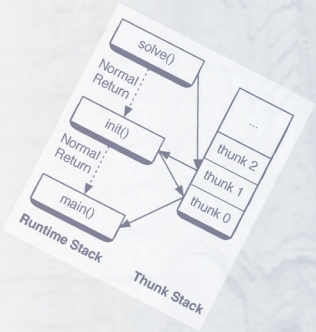
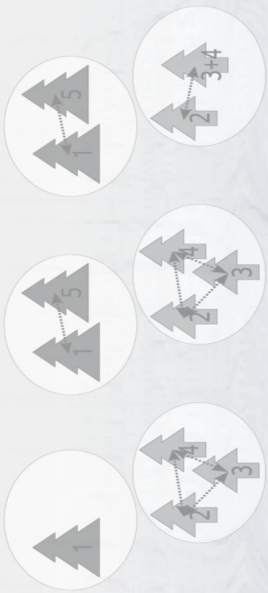
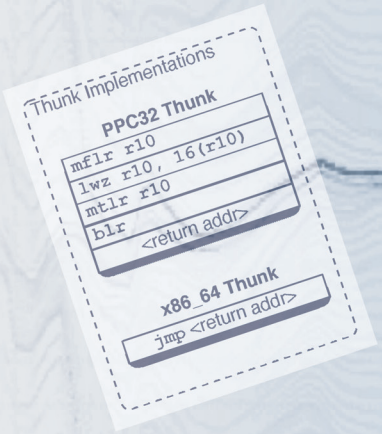
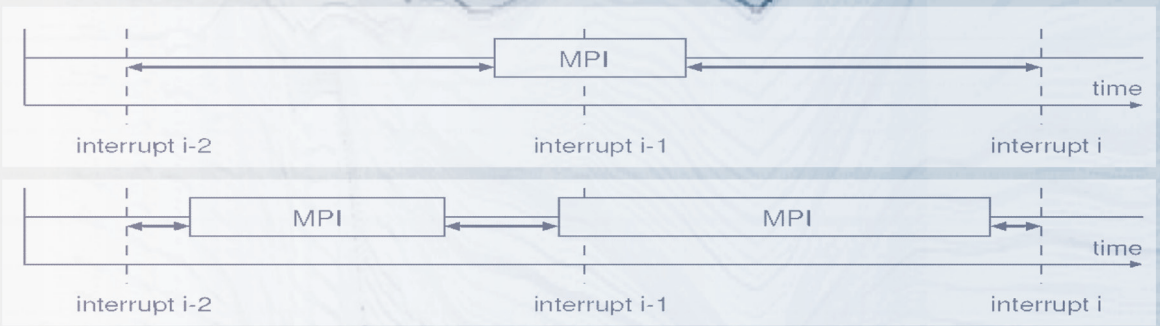








Capturing Parallel Performance Dynamics

Zoltán Péter Szebenyi

Member of the Helmholtz Association

JÜLICH
FORSCHUNGSZENTRUM

Schriften des Forschungszentrums Jülich
IAS Series Volume 12

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Capturing Parallel Performance Dynamics

Zoltán Péter Szebenyi

Schriften des Forschungszentrums Jülich
IAS Series Volume 12

ISSN 1868-8489 ISBN 978-3-89336-798-6

Bibliographic information published by the Deutsche Nationalbibliothek.
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Publisher and Distributor:	Forschungszentrum Jülich GmbH Zentralbibliothek 52425 Jülich Phone +49 (0) 24 61 61-53 68 · Fax +49 (0) 24 61 61-61 03 e-mail: zb-publikation@fz-juelich.de Internet: http://www.fz-juelich.de/zb
Cover Design:	Grafische Medien, Forschungszentrum Jülich GmbH
Printer:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2012

Schriften des Forschungszentrums Jülich
IAS Series Volume 12

D 82 (Diss., RWTH Aachen University, 2012)

ISSN 1868-8489
ISBN 978-3-89336-798-6

The complete volume is freely available on the Internet on the Jülicher Open Access Server (JUWEL) at http://www.fz-juelich.de/zb/juwel

Persistent Identifier: urn:nbn:de:0001-2012062204
Resolving URL: http://www.persistent-identifier.de/?link=610

Capturing Parallel Performance Dynamics – Abstract

Supercomputers play a key role in countless areas of science and engineering, enabling the development of new insights and technological advances never possible before. The strategic importance and ever-growing complexity of the efficient usage of supercomputing resources makes application performance analysis invaluable for the development of parallel codes. Runtime call-path profiling is a conventional, well-known method used for collecting summary statistics of an execution such as the time spent in different call paths of the code. However, these kinds of measurements only give the user a summary overview of the entire execution, without regard to changes in performance behavior over time. The possible causes of temporal changes are quite numerous, ranging from adaptive workload balancing through periodically executed extra work or distinct computational phases to system noise. As present day scientific applications tend to be run for extended periods of time, understanding the patterns and trends in the performance data along the time axis becomes crucial.

A straightforward approach is profiling every iteration of the main loop separately. As shown by our analysis of a representative set of scientific codes, such measurements provide a wealth of new data that often leads to invaluable new insights. However, the introduction of the time dimension makes the amount of data collected proportional to the number of iterations, and memory usage and file sizes grow considerably. To counter this problem, a low-overhead online compression algorithm was developed that requires only a fraction of the memory and file sizes needed for an uncompressed measurement. By exploiting similarities between different iterations, the lossy compression algorithm allows all the relevant temporal patterns of the performance behavior to be reconstructed.

While standard, direct instrumentation, which is assumed by the initial version of the compression algorithm, results in fairly low overhead with many scientific codes, in some cases the high frequency of events (e.g., tiny C++ member function calls) makes such measurements impractical. To overcome this problem, a sampling-based methodology could be used instead, where the amount of measurement overhead becomes a function of the sampling frequency, independent of the function-call frequency. However, sampling alone is insufficient for our purposes, as it does not provide access to the communication metrics the compression algorithm heavily depends on. Therefore, a hybrid solution was developed that seamlessly integrates both types of measurement techniques in a single unified measurement, using direct instrumentation for message passing constructs, while sampling the rest of the code. Finally, the compression algorithm was adapted to the hybrid profiling approach, avoiding the overhead of pure direct instrumentation.

Evaluation of the above methodologies shows that our semantics-based compression algorithm provides a very good approximation of the original data with very little measurement dilation, while the hybrid combination of sampling and direct instrumentation fulfills its purpose by showing the expected reduction of measurement dilation in cases unsuitable for direct instrumentation. Beyond testing with standardized benchmark suites, the usefulness of these techniques was demonstrated by their key role in gaining important new insights into the performance characteristics of real-world applications.

Erfassung paralleler Leistungsdynamik – Zusammenfassung

Hochleistungsrechner spielen eine Schlüsselrolle in einer Vielzahl von Disziplinen in Wissenschaft und Technik. Sie ermöglichen die Entwicklung neuer Erkenntnisse und technologischen Fortschritt in nie da gewesenem Maße. Der strategische Stellenwert von Hochleistungsrechnern bei gleichzeitig stetigem Anstieg der Nutzungskomplexität macht die Leistungsanalyse paralleler Anwendungen unverzichtbar für deren Entwicklung. Callpath-Profiling ist eine gängige Methode, um Laufzeitstatistiken, wie zum Beispiel die Zeit, welche die Anwendung in verschiedenen Aufrufpfaden verbringt, zu ermitteln. Diese Art der Messung gibt dem Entwickler jedoch nur einen Überblick über die gesamte Laufzeit der Anwendung, ohne zeitliche Veränderungen im Programmverhalten zu berücksichtigen. Die möglichen Ursachen solcher zeitlichen Variationen können vielschichtig sien, von adaptiven Lastumverteilungen über periodisch auftretende zusätzliche Funktionsaufrufe und spezielle Berechnungsphasen bis hin zu Systemrauschen. Da heutige wissenschaftliche Anwendungen aber oft über längere Zeiträume ausgeführt werden, ist es jedoch unerlässlich, die zeitlichen Muster und Tendenzen in den Leistungsdaten zu erfassen und zu verstehen.

Um dies zu bewerkstelligen, wird in einem einfachen Ansatz für jede Iteration der Hauptschleife ein separates Profil erstellt. Wie durch eine Analyse von repräsentativen wissenschaftlichen Simulationen gezeigt, liefern solche Messungen eine Fülle neuer Daten und wertvolle Einsichten in das Laufzeitverhalten. Durch die Einführung der Zeitdimension wächst jedoch die Datenmenge mit der Anzahl der Iterationen, wodurch Hauptspeicherbedarf und Dateigrößen beträchtlich ansteigen. Um diesem Problem zu begegnen, wurde in dieser Arbeit ein effizientes Online-Kompressionsverfahren mit geringem Laufzeitoverhead entwickelt, das den für die Messdaten benötigten Platz auf einen Bruchteil reduziert. Durch Ausnutzung von Ähnlichkeiten zwischen verschiedenen Iterationen gestattet der verlustbehaftete Kompressionsalgorithmus die Rekonstruktion aller relevanten zeitlichen Muster.

Obwohl vollständige direkte Instrumentierung, welche dem ursprünglichen Kompressionsalgorithmus zugrunde liegt, bei vielen wissenschaftlichen Simulationen mit vertretbarer Laufzeitdilation genutzt werden kann, gibt es Fälle, bei denen die hohe Frequenz von Messpunkten, wie zum Beispiel Aufrufe von in C++ üblichen extrem kurzen Objektmethoden, dies unmöglich macht. Um diesem Problem zu begegnen, wird normalerweise auf Sampling ausgewichen, dessen Overhead proportional zur Samplingfrequenz wächst, jedoch unabhängig von der Frequenz der Messpunkte bzw. Funktionsaufrufe ist. Sampling allein ist allerdings nicht adäquat für das obige Kompressionsverfahren, da es keinen Zugriff auf die Kommunikationsmetriken gestattet, anhand derer die Ähnlichkeit von Iterationen festgestellt wird. Daher wurde eine Hybridlösung entwickelt, die beide Messmethoden nahtlos miteinander in einer einzigen Messung verbindet. Dabei wird direkte Instrumentierung für die Erfassung der Kommunikationsmetriken und Sampling für den Rest der Anwendung genutzt. Abschließend wurde der Kompressionsalgorithmus an diesen Hybridansatz angepasst, um so den Overhead reiner direkter Instrumentierung zu umgehen.

Die Auswertung hat gezeigt, dass der die Semantik der Daten ausnutzende Kompressionsalgorithmus eine sehr gute Näherung der Originalmessungen mit sehr geringer Laufzeitdilatation liefert, während die hybride Nutzung von direkter Instrumentierung in Kombination mit Sampling den Messoverhead in denjenigen Fällen reduziert, die für direkte Instrumentierung allein nicht geeignet sind. Neben Tests mit standardisierten Benchmarks haben die hier vorgestellten Methoden bei der Gewinnung von wichtigen Einsichten in die Leistungscharakteristiken von echten Simulationsanwendungen ihren Nutzen erwiesen.

Acknowledgements

I would like to thank Prof. Dr. Marek Behr, Scientific Director of AICES and Vice-President of the German Research School for Simulation Sciences; and Prof. Dr. Dr. Thomas Lippert, Director of the Jülich Supercomputing Centre, for giving me the opportunity to carry out my Ph.D. project in these excellent research environments.

The enthusiastic supervision and guidance of Prof. Dr. Felix Wolf, with his professionalism and extreme attention to detail were instrumental to the success of this project, and I owe him a deep debt of gratitude. Moreover, I thank Prof. Dr. Michael Gerndt and Prof. Dr. Uwe Neumann for serving as second referees, and Prof. Dr. Erika Ábrahám, Prof. Dr. Leif Kobbelt and Prof. Dr. Thomas Seidl for serving on my examination committee.

I am also grateful to Dr. Bronis R. de Supinski and Dr. Martin Schulz for hosting me at the Lawrence Livermore National Laboratory. My lively and thought-inspiring discussions with them, along with Dr. Todd Gamblin and other scientists at the lab lead to important new ideas in this thesis project.

Thanks are due to colleagues at the Jülich Supercomputing Centre and the German Research School for Simulation Sciences for numerous stimulating discussions, help with experimental setup, and general advice. I would especially like to acknowledge the invaluable assistance of Dr. Brian Wylie, who always had time to discuss results and provide me with his expert and deeply insightful advice, hugely contributing to my professional development. Not to be forgotten, thanks are also due to my Ph.D. colleagues for their help and support in myriad situations.

I would like to especially thank Lászlóné Szegfű, Zsolt Fodor, Gábor Nikolényi, Prof. Dr. Gyula Horváth, Prof. Dr. Mihály Bohus, Prof. Dr. Tibor Csendes and Prof. Dr. János Csirik for being defining characters in my education, who gave me their time, support and wisdom far beyond their duties as teachers and professors. I would have never made it this far without them. A great thanks goes also to Sifu Yücel Arslan and Sifu Hubert Werner for their wisdom and friendship.

I am grateful to all my friends for their support and friendship in various phases in my life, who are too numerous to mention here. I am also grateful to the Hungarian communities in Aachen and Livermore, who gave me moral and material support and provided me with a warm community to belong to. Last but by no means least, I shall be forever indebted to my parents and family for providing me with a stable background and absolute support in every situation.

Zoltán Péter Szebenyi
June 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The single word that characterizes the field of supercomputing is "growth". Everything grows as the years pass by: number of processors, cores, threads; input datasets; complexity of both software and hardware; the user base; and last but not least, the electricity bills. Parallel computing is growing, and so is its already significant importance. Tools that can effectively help in making software use the available parallel hardware performance efficiently are more valuable than ever. The goal of this project was to advance the state-of-the-art in the field of parallel performance analysis by providing deeper insight into the time-dependent performance behavior of high performance computing applications. To get a good understanding of how and what we did to reach this goal, we have to start at the basics.

1.1 Supercomputers

The term *supercomputer* can be applied to a variety of different kinds of machines, many of them being quite distant relatives. Old trends fade away and new ones appear on the horizon. The usage of accelerator blades, FPGAs (field-programmable gate arrays), GPUs (graphics processing units) are current trends, some of which might gain more ground in the future. In this work, we are concentrating on two classes of mainstream supercomputers, Linux clusters and specialized large machines, in particular IBM's Blue Gene/P solution. Linux clusters are the standard choices for universities [72] and similar institutions [49] that have to support a variety of users, as they are powerful all-around machines built from commodity parts occasionally combined with some specialized networking hardware. They tend to make use of the same line of processors as high-end desktop PCs, which often means some x86_64 solution these days.

Blue Gene [44] on the other hand is a thoroughbred supercomputer, and it is only usable by developers who have made a good job parallelizing their code already. Blue Gene started off as an exotic experiment like many others, and it turned out to be a huge success, with several installations among the top of both the Top 500 [2] and Green 500 [1] lists, which means top performance combined with extreme energy efficiency. By now it constitutes its own category in supercomputing with its unorthodox solution of sacrificing single core performance at a clock speed of 850MHz in exchange for lower voltage, compact packaging, easier cooling, lower energy consumption, better memory/processor clock ratio and lots of processor cores. And to make this all work together at top performance, it has multiple interconnect networks,

all optimized for their specialized tasks: a three-dimensional torus network for point-to-point communication, specialized barrier network for extra quick `MPI_Barrier` calls, and a tree network for MPI reductions. The operating system that runs on those cores is a minimal Linux kernel that runs a single process only. This simplicity on the software side and the specialized networking hardware provide an environment where system noise is virtually non-existent. Ideal for developers interested in reproducible performance and performance tuning.

Large supercomputing centers often have multiple supercomputers with different capabilities to be able to serve all possible user needs. Their largest machines, which are often specialized systems like Blue Genes are called *capability systems*, while the smaller, more standard all-around systems are called *capacity systems*. The capability system is used by a limited number of high priority, highly scalable codes, capable of making good use of the extreme characteristics of such a system, while the capacity system is used to serve the needs of all other projects. It is interesting to note that the naming scheme would sometimes also work the other way around, as capability systems often have a higher capacity while capacity systems often have more capable individual processors and compute nodes.

1.2 Parallel programming

Most scientific codes are still serial, or start off as serial projects and are only parallelized later on through a series of patches and hacks, which leads to suboptimal solutions. Of course many codes are still in a kind of prototype phase, implemented in e.g. Matlab. They still have to make the most important and perhaps hardest step towards supercomputing, switching to a programming language capable of driving the hardware at its full performance, which usually means C/C++ or Fortran. It is possible to get good performance with Matlab, especially when the problem maps well to the available libraries, but the specialized hardware and extreme parallelism of today's supercomputers often requires the usage of lower-level languages.

Once the code is in C/C++/Fortran, people are happy with the performance gained from switching programming languages, but at some point they eventually realize that they need more. They want to process more data, they want quicker answers. At this point, the quickest and easiest step towards parallelization is usually via *OpenMP* [71], which has the added benefit that parallelization can be done incrementally, in arbitrarily small steps. The user inserts compiler directives into the code that help the compiler parallelize the marked parts of the code. It is usually enough to insert these directives for the most important core solver loops to get a performance boost from the machine's multithreading capabilities on SMP (symmetric multiprocessing) compute nodes. The drawback of OpenMP is that it is limited to the number of cores on a single compute node that can be used simultaneously without oversubscription. This usually means around 8 to 16 threads, and even when using tricks like hyper-threading typically not more than 256 threads. This is due to the fact that OpenMP requires *shared memory*, which is only available inside a single compute node, with the above mentioned limited number of cores. Of course there are special machines with very large processor counts sharing the same memory address space using solutions like ccNUMA (cache-coherent Non-Uniform Memory Access) architecture, or software-

based virtual shared memory solutions [76], but they are still the exceptions, not the typical, mainstream supercomputers.

To gain access to more parallelism, developers have to embrace the idea of *distributed-memory programming*, with all its challenges. Distributed-memory machines simply mean lots of shared-memory compute nodes connected through a high performance network. The problem here is that it is not possible for the compiler to take care of data transfers automatically any more. The user has to specify all communications and synchronizations between the processes on the compute nodes explicitly, for which the standard tool is *MPI* [60], or the *Message Passing Interface*. It is a standardized API provided on every supercomputer — often by fine-tuning an open-source implementation for the given architecture — and provides basic as well as more advanced communication and synchronization capabilities. There are two extreme cases of parallelizing an already existing serial code using MPI: the developers may be lucky enough that the parallelization of their existing code is feasible with relatively little effort, or they may have to start over from scratch. Of course there is a whole spectrum of options, such as parallelizing only certain parts of the code. Still, both extremes have their pros and cons. Once they overcome this challenge, one or the other way, and their code scales reasonably well up to a few hundred cores, they can be considered expert users. They are probably eligible for larger allocations on capacity systems with their codes. To get even further, and become elite users, they can take advantage of the help of parallel performance experts, or tune their code themselves until it scales well to thousands, or tens of thousands of cores. At that point, they become eligible for an allocation on a capability system. Of course taking advantage of the full machine available there, perhaps hundreds of thousands of cores requires lots of additional effort, often completely replacing certain algorithms with more scalable ones, but it is worth noting how much progress we made already from the original serial prototype code in Matlab.

At this point, depending on their codes, users may choose to switch from MPI to a hybrid combination of MPI and OpenMP, which has the potential for additional performance gains by using OpenMP for intra-node synchronization while MPI is used for inter-node communication and synchronization only. Using shared memory for intra-node communication has the advantage of (i) reduced memory usage (e.g. for ghost cells or communication buffers), (ii) less data movement necessary between these cores and (iii) the number of MPI links between nodes can be kept at a minimum for improved scalability.

1.3 Serial performance tools

The most well known performance analysis tools are probably the classic serial tools, such as *gprof* [18] or *Intel VTune* [46] (which also has parallel components now). The most basic technique employed by these tools is sampling-based profiling, which means setting a timer to generate an interrupt at predetermined time or hardware counter intervals, reading the execution state at every interrupt, and aggregating and displaying overview statistics in post-mortem analysis. These tools might have a myriad of more advanced features, including compiler-based instrumentation, which enables exact, deterministic measurements; and call-stack unwinding, which provides information about complete call paths in sampling-based measurements, at the price of potentially higher measurement dilation. We differentiate

between two main kinds of profiling: *flat profiling*, where the tool provides performance statistics broken down by functions, and *call-path profiling*, where the complete call path to each function call is also stored.

Call-path profiling [35, 7, 16, 58, 9], which aggregates performance metrics across the entire execution broken down by call path, is a widely-used method of linking a performance problem to the context in which it occurs. Especially when investigating the use of library functions that may be used at multiple places in the source with different parameters, call-path information can be critical in tracking performance problems back to their roots in the user code. For example, during the analysis of MPI programs, call-path information is often essential to decide where in the program a communication bottleneck occurs. However, the context expressed in the call path usually reveals little about the temporal evolution of a performance phenomenon. As a result, important insights as to how and why certain behaviors develop may be lost.

1.4 Parallel performance measurement and analysis

Effectively harnessing the many-fold parallelism available on modern supercomputers becomes increasingly challenging. In particular, meeting the performance expectations for programs running on today's complex hardware can require significant effort. We can reduce this effort through appropriate performance-analysis technology, such as performance profiles that measure the execution time spent in different parts of the program. State-of-the-art parallel profilers, such as HPCToolkit [3] and TAU [79], further reduce the effort through context-sensitive analysis that differentiates not only between different functions, but also between the call paths leading to those functions, which delineate the performance phenomena more precisely. In addition to time and hardware counters, parallel profilers may also acquire parallelization metrics such as message counts and the communication volume.

Parallel performance tools are usually more complex as they have to deal with data collected by potentially thousands of processes and threads, while collecting more data per process than serial performance tools. Serial tools concentrate on measuring time and hardware counters, while the main focus for parallel performance tools is largely shifted towards communication and synchronization performance. This is due to the fact that it does not matter how quickly code could potentially run when it is not running. And that is exactly what happens in processor cores while waiting for communications to finish: they are doing nothing productive. When a hundred thousand processes do nothing most of the execution time, because they have to wait on a single process that has more workload, it can quickly become extremely expensive. These kinds of problems are inevitably becoming increasingly common just based on factors such as system noise as we move on towards computers with millions or billions of cores, as shown by [41]. Still, keeping this problem to the very minimum by understanding and fine-tuning the communication performance is necessary in order to minimize wastage of resources.

Most importantly, the user might be wasting huge resources without even realizing it. There is no easy way to tell if a code is making efficient use of the hardware or wasting 95% of the

potential computing power, without taking performance measurements using tools. Of course taking measurements at different scales and drawing scalability graphs gives an overall idea. There are two main modes of scaling applications: in *weak scaling* the problem size is scaled proportionally to the process count, leading to equal execution times in the optimal case, while in *strong scaling* the same problem is solved using more processes, with the intention of calculating the solution more quickly. All the test cases used in this thesis are using strong scaling. If the scalability graph looks bad, there is no chance that the application is efficient at scale, but even if the graph shows linear speedup, it might still be wasting half of the time waiting in communications. The only way to a definite answer is to measure using a parallel performance tool.

Clearly, drawing a simple scalability graph is just a first step in predicting application performance at different scales or on different machines, there is a separate discipline called *performance modeling*, which builds an elaborate model of the application performance, often using detailed knowledge of the underlying algorithms and the architecture of the machine. Performance modeling is usually applied to applications of high importance, as it requires significant amounts of manual labor from specialized domain experts. Parallel performance tools are also invaluable in the performance modeling process, both in the creation and the validation of the performance model [50, 82, 40].

1.4.1 MPI measurements and metrics

Parallel performance tools measure specialized metrics beyond execution time. They use the PMPI profiling interface [60], which is based on weak symbols pointing to the actual implementation functions. By overriding the weak symbols with wrapper functions (e.g. by library preloading), the tools get information about every MPI call, including the complete argument list. The most important metrics obtained in this way include *MPI communication time*, further broken down to *Point-to-point communication time*, *Collective communication time* and *Collective synchronization time*; and some count-based metrics, such as *Collective communication count* or *Bytes transferred*, to name a few examples. Some MPI applications also use one-sided *Remote Memory Access (RMA)* operations or MPI file I/O and additional metrics may be provided for them.

1.4.2 Profiling vs. tracing

Parallel performance analysis tools can be distinguished by the way they collect and store data during measurement. The main categories are profiling and tracing tools, or combinations thereof. Profiling tools accumulate aggregate information about the application's execution, immediately extracting the most important statistics from events on the fly. In contrast, tracing tools collect every event into a large buffer and provide this very detailed data for post-mortem analysis and/or visualization. These measurements can provide much more detailed analysis than profiling tools, but they are generally harder to use properly due to limited buffer space capacity and prohibitive file I/O costs. To summarize the differences:

- *Profiling* is used to get a compact summary of the execution characteristics.
- *Tracing* is used to record all relevant performance events as they occur for detailed post-mortem analysis.

Especially on present day supercomputers, memory space is becoming an increasingly scarce resource. Experienced performance analysts would first take uninstrumented measurements for reference at different scales, then take a first profile measurement and use it to fine-tune parameters for subsequent measurements. Once the first low-overhead profiling measurements are complete, they can decide if subsequent tracing measurements are necessary and/or feasible.

TAU [79] is a typical profiling tool with an outstandingly large feature set. It is known for early adaptation of new trends such as supporting JAVA, Python or GPU-based systems.

Vampir [68] is a trace collector with a powerful commercial trace visualizer component. It allows the user to zoom into the collected event trace data and see how long certain function calls or communications took, visualized using parallel timelines, as shown in Figures 1.2 & 1.3. Messages between processes are shown using arrows between the send and receive events in a quite intuitive way. Vampir is a great tool for exploring the details of certain performance problems, but the performance analyst can quickly become a bottleneck in the analysis process, as it is beyond human limits to grasp the meaning of visualizations depicting thousands of processes each with millions of events.

This problem is partially resolved by the *Scalasca* profiling and tracing tool, which gets rid of this performance bottleneck by automating the trace analysis. Parallel traces contain such a huge amount of data that processing them serially is impractical even for a computer. This is why Scalasca uses a parallel trace analysis that runs on the same number of cores as the original application, thereby solving the scalability problem introduced by growing process counts. As the analysis is replay-based, the analysis tool is roughly as scalable as the application being analyzed, although it often runs somewhat quicker as it only has to replay the communications but not the computation. It looks for higher level communication inefficiency patterns, a typical example of which is *Late sender time*. It means that a process started waiting on a message, but the timestamps in the trace of the sender process indicate that the corresponding send call was not yet initiated.

As Scalasca is the tool we extended with additional features during this project, we introduce it in somewhat more detail.

1.4.3 Scalasca toolset

Scalasca is an open-source toolset for scalable performance analysis of large-scale parallel applications [90, 74]. It includes integrated runtime measurement summarization and event tracing [94] with automatic trace analysis based on parallel replay [29], to ensure scalability for long-running and highly-parallel MPI, OpenMP and hybrid applications.

When the Scalasca instrumenter is prepended to each of the application's compile and link commands, it produces fully-instrumented executables, exploiting the standard PMPI library interposition interface, and function entry and exit instrumentation capabilities provided by

most modern compilers. It is important to note that compiler-based instrumentation has a potential for inhibiting certain compiler-based optimizations. A source preprocessor is also provided for OpenMP pragma/directive and annotated region instrumentation (though not used in this work). Manual annotation of significant code regions (e.g., initialization, phases) can also be done using a macro-based user API.

Scalasca measurement collection and analysis is performed by a workflow manager that is prefixed to the normal application-execution command line, whether part of a batch script or interactive run invocation. Performance metrics such as execution time, message statistics, and hardware counters are aggregated individually for every thread and call path encountered during the entire program execution, creating a call-path profile, which merges the information gained from the PMPI wrappers about the MPI calls and from user-code instrumentation about the user call paths. Experiments with an instrumented executable can be configured to collect runtime profiles and/or event traces (optionally including hardware counters [93]), with the latter automatically analyzed with the same number of processes as used for measurement. Whereas the former are needed to identify the most resource-intensive call paths in the program, the latter can be used to identify call paths that exhibit a significant fraction of idle time. The result of such a trace analysis is therefore similar in structure to a runtime call-path profile but enriched with higher-level communication and synchronization inefficiency metrics. Both summary and trace analyses are generated in the same profile format, which can be interactively explored with the Scalasca analysis report explorer GUI (shown in Figure 1.4). Command-line tools are also provided for processing analysis reports, for example to compare measurements or produce *filters* for subsequent measurements (which we will describe later in more detail). An overview of the Scalasca experiment workflow is shown in Figure 1.1.

The metrics collected during a profiling measurement of MPI applications are listed in Table 1.1 (on page 16), with indentation representing the hierarchical relationship between them. They are divided into a subset $M = \{M_1, \ldots, M_m\}$ that is directly measured and another subset $D = \{D_1, \ldots, D_d\}$ derived by later analysis (set in italic). The metrics shown with white background are the most significant ones, whereas the metrics with gray background are either not relevant for the techniques introduced here, or not appearing in our measurements. For example, initialization and finalization of both the measurement system and MPI are typically one-off expenses outside the primary computational loop that we are trying to characterize. Optional hardware counter metrics and specific MPI metrics related to file I/O, collective communication, etc., may not exist or be entirely zero-valued in certain measurements. OpenMP-related metrics are also not used here, as the focus is on pure MPI-based applications in this document.

Scalasca defines call paths as lists of visited regions (usually starting from the `main` function) and maintained in a tree data structure. Thus, a new call path is specified as an extension of a previously defined call path to the new terminal region. When a region is entered from the current call path, any child call path and its siblings are checked to determine whether they match the new call path, and if not a new call path is created and appropriately linked (to both parent and last sibling). Exiting a region is then straightforward as the new call path is the current call path's parent call path. When execution is complete, a full set of locally-executed call paths are defined, which are merged into a global set during program finalization, serving as a basis for later call-tree visualizations.

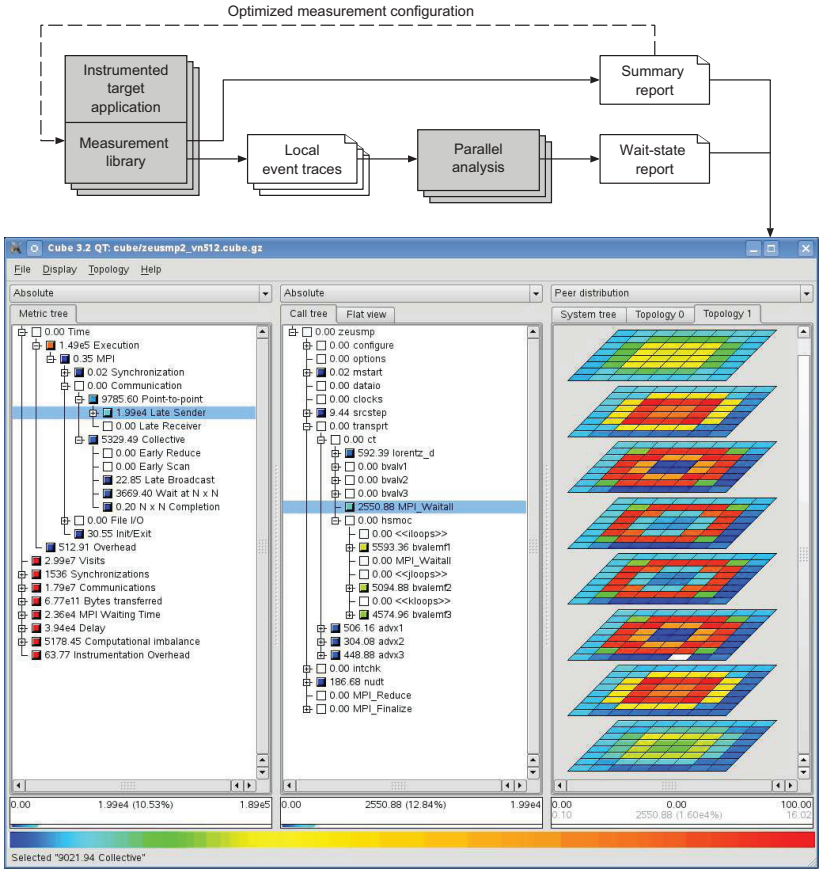


Figure 1.1: *Schematic overview of the performance data flow in Scalasca. Grey rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files running or being processed in parallel. The GUI shows the distribution of performance metrics (left pane) across the call tree (middle pane) and the process topology (right pane).*

1.5 Motivation: Sweep3D

In our study [91] of the execution performance of the ASC *Sweep3D* [56] application on the Blue Gene/P system in Jülich, we investigated a serious computational imbalance that severely impacted scalability. The analysis of *Sweep3D* uncovered several interacting performance patterns. Each iteration consists of several "sweeps", each one sending data in one of the eight possible diagonal directions in the application's three-dimensional coordinate system. These sweeps introduce a communication imbalance, as the processes receive data from their neighbors in the order dictated by the current sweep direction. Furthermore, after four sweeps, the fifth one goes in the exact opposite direction of the fourth sweep, which means that the complete "sweep pipeline" has to be empty before the fifth sweep can start, it is not possible to overlap those sweeps. Figures 1.2 & 1.3 show a 1024-process trace of a *Sweep3D* execution in the Vampir viewer. Figure 1.2 gives an overview of the complete execution, while Figure 1.3 zooms in on a single iteration, iteration 8 (of twelve).

In the top part of the figure we see the timelines of a small subset of the processes, where red means communication (MPI) and turquoise means computation. The above mentioned imbalance caused by the sweep calls is clearly visible in these timelines. In the center of the figure, the timeline view for process 0 shows two sweep function calls in this iteration. This is because four logical sweeps are combined into a single sweep function call, so an iteration contains only two sweep function calls and not eight. The lower part of the figure shows the communication matrix. At first glance it might not look very informative, but it actually gives us a lot of interesting information, including a first indication of another important performance pattern. The process range between 608 and 639 is one column of the two-dimensional process topology of the application which is responsible for computations in a single plane in the three-dimensional logical coordinate system. The communication matrix tells us that communication times are significantly higher in the middle part of this process range, indicating that there might be an imbalance between the processes in the middle and the processes on the edges.

Indeed, a look at the application's two-dimensional grid topology in Figure 1.4 confirms this, showing a clearly visible, well-defined imbalance pattern differentiating the a rectangular region in the middle of the topology from the rest of the processes. Using manual source-code instrumentation annotations we were able to isolate the origin to a serious workload imbalance that only manifests in certain solver iterations. After adding macros to isolate specific regions of the core `sweep` function, we found that the code region where corrective 'fixups' are applied is responsible for the imbalance. A lower number of fixup iterations was necessary on the processes in an interior rectangular block of the logical topology than in other processes, leading to the imbalance. The pattern is shown in the right panel of the screenshot in Figure 1.4. Since input configuration specifies that fixup operations are only applied after the seventh iteration, the major solver iteration loop in the `inner` function was annotated with increasing values of the iteration variable `its`, each time defining a new region labeled with the corresponding value of `its`, as seen in the call-tree panel (middle) of the screenshot in Figure 1.4. The topology display in the screenshot provides a summary of the *Exclusive execution time* in all iterations, while the bottom row shows the evolution of the same metric

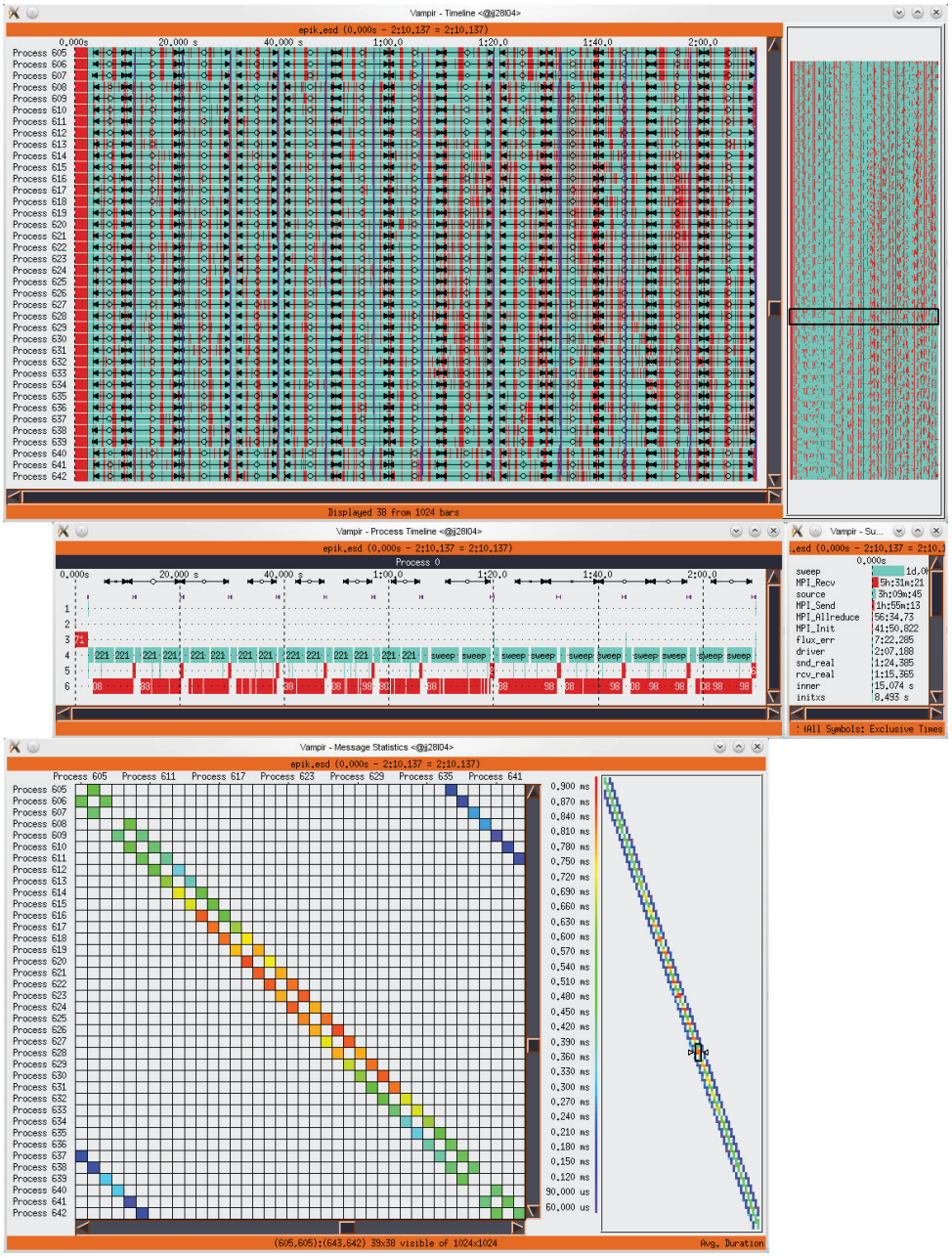


Figure 1.2: *Vampir interactive analysis and visualization of an entire trace of a* Sweep3D *execution on BG/P with 1024 MPI processes, showing master timeline view (top) with 'thumbnail' navigation panel, process timeline view (center) for process 0 showing the 12 iterations of double* `sweep` *calls and execution time summary profile, and communication matrix (bottom) showing average duration of nearest-neighbor communication.*

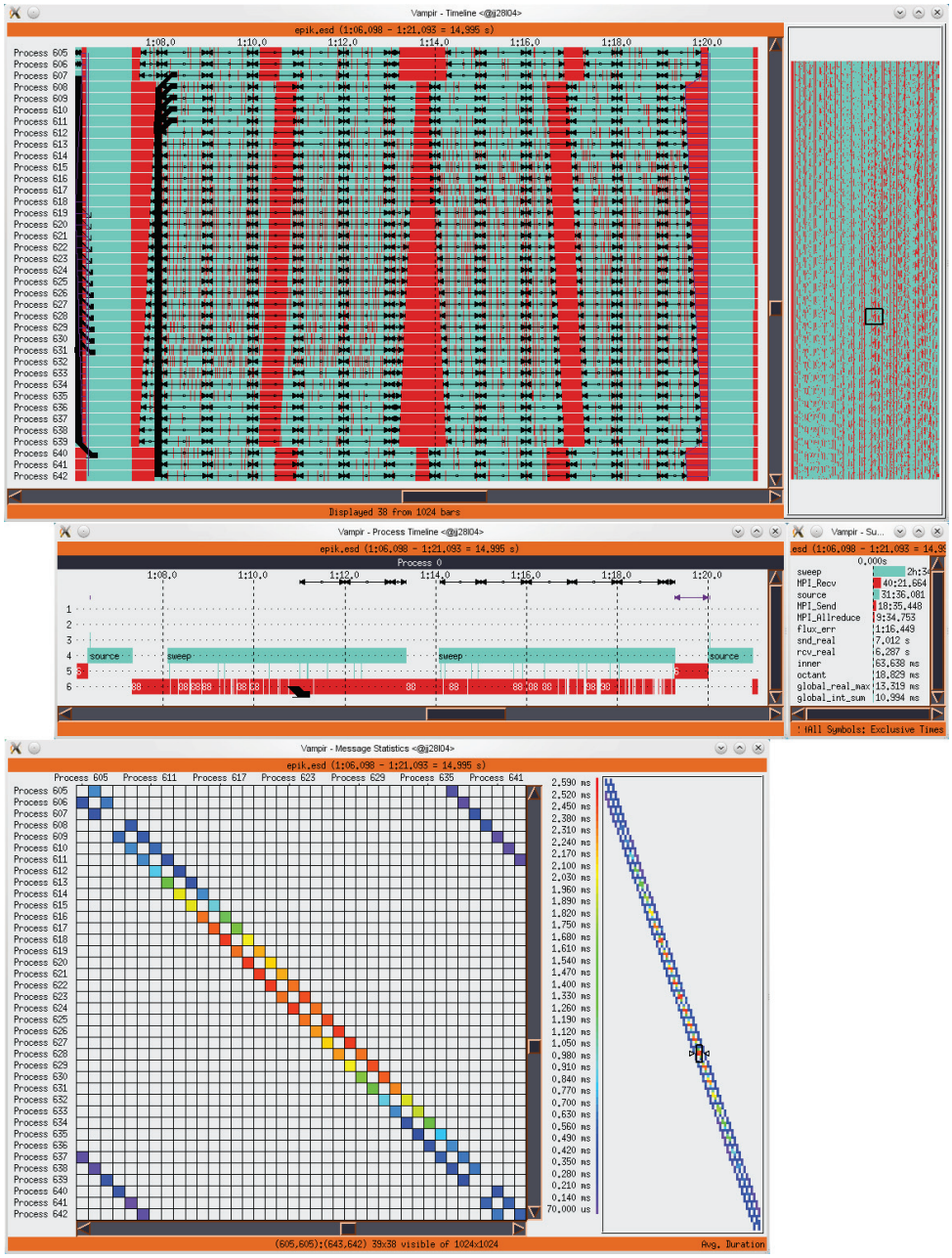


Figure 1.3: *Vampir interactive analysis and visualization of imbalanced iteration 8 in trace of Sweep3D execution on BG/P with 1024 MPI processes, showing master timeline view (top) with 'thumbnail' navigation panel, process timeline view (center) for process 0 showing the double* `sweep` *calls and execution time summary profile, and communication matrix (bottom) showing average duration of nearest-neighbor communication.*

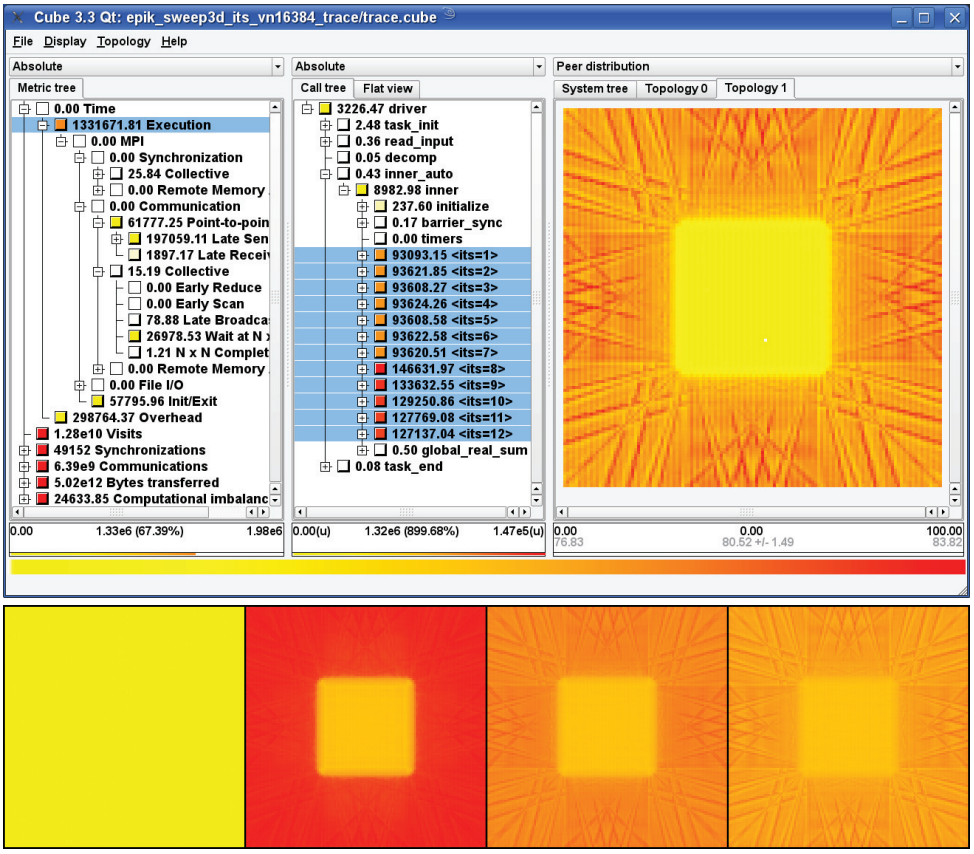
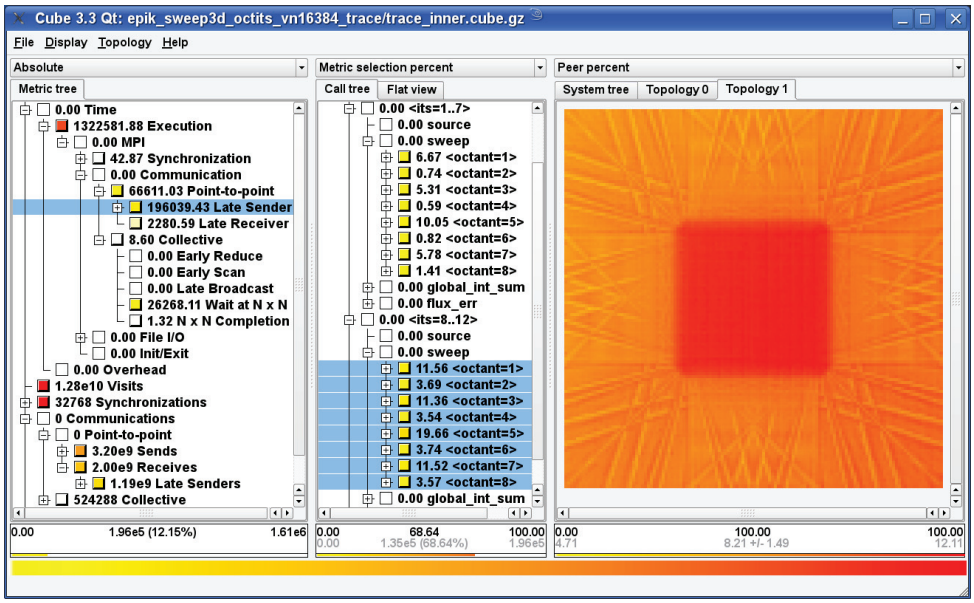


Figure 1.4: *Execution time metric variation by iteration in a 16,384-process Sweep3D execution on the Blue Gene/P (top, center), and computational imbalance evolution shown using the logical topology for iterations 7 to 10 (bottom).*

in iterations 7-10. Everything is balanced up to iteration seven, and the imbalance kicks in at iteration eight, where the fixup operation is applied for the first time.

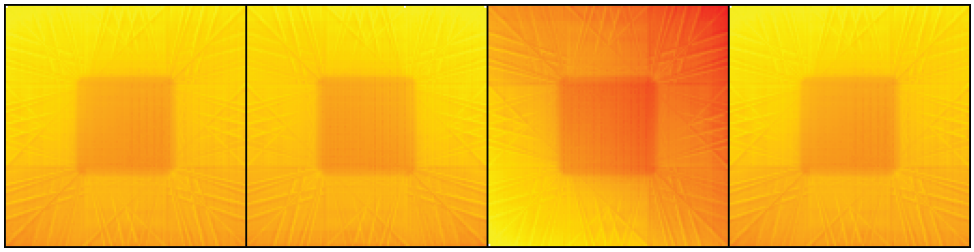
Drilling down deeper, more manual instrumentation was applied to distinguish the sweep octants of the main solver core, that is, the eight subsequent sweep directions executed in the core. Figure 1.5 shows the distribution of the *Late Sender* waiting time as a complement to the distribution of pure computation time arising from the fixup calculations seen in Figure 1.4. Under the GUI screenshot, the top row shows how the different octant sweeps cause gradual changes in *Late Sender* waiting time corresponding to the specific sweep directions in iterations with no fixup operations, while in the bottom row this pattern is clearly superimposed with the imbalance pattern caused by the fixups, as shown in Figure 1.4.

It is clear that this kind of detailed information about specific iterations and sub-phases of those iterations is generally useful and provides insight that would otherwise not be possible,



Iterations without fixups:

Iterations with fixups:



octant=1+2 octant=3+4 octant=5+6 octant=7+8

Figure 1.5: *MPI Late sender time metric variation by sweep octant in a 16,384-process Sweep3D execution on the Blue Gene/P for initial 7 non-fixup and subsequent 5 fixup iterations (top) and waiting time distributions for the octant pairs 1+2, 3+4, 5+6, 7+8 in computationally balanced non-fixup (middle) and imbalanced fixup (bottom) iterations.*

not only in this case, but also in many other applications. In general, as numerical simulations model the temporal evolution of a system, their progress usually occurs via a series of discrete points in time. According to this iterative nature, the core of such an application is typically a loop that advances the simulated time step by step — the entire loop often preceded by initialization and concluded by finalization procedures. However, the manual instrumentation necessary to define separate regions for each iteration is too inconvenient for performance analysts to follow this method regularly, even for a dozen iterations as seen in this example. This makes a clear case for the necessity of support for iteration and phase-instrumentation capabilities in our measurement system, which was implemented as part of this thesis project. This now provides an easy-to-use API for applying such iteration instrumentation with as little manual work as possible.

But creating the iteration instrumentation API was just the first step. It was not the first such API ever designed, and certainly also not the last. The specific goal for its design was to provide a wide range of possibilities for further experimentation on how to collect time-series call path profiles in the most efficient way. So we started experimenting. Our studies of the SPEC MPI2007 benchmark suite and other applications gave us reassurance and motivation for digging deeper into the topic, proving that this simple technique can uncover a wealth of new and relevant data about the time-dependent performance behavior of HPC applications. As we will see in Chapter 3, there is a wide range of non-trivial performance patterns that can only be observed by taking the time dimension into account.

However, the new technique presented us with new challenges along with the new opportunities. The memory and disk space requirements of time-series profiling measurements grew linearly with the number of iterations, which could easily reach thousands or more in many cases. In order to reduce these requirements, a new on-line, lossy, clustering-based compression algorithm was developed, taking advantage of the inherent redundancy in the generated time-series profiles. While the compression is lossy, meaning that not all metric values can be reconstructed exactly, our evaluations show that the algorithm provides a good (and configurable) tradeoff between compression quality and resource utilization. Moreover, the algorithm provides certain problem-specific guarantees, the most important being that the call tree structure of every iteration can be reconstructed exactly, even if different iterations sometimes have different call trees.

Another important issue we needed to resolve was the excessive measurement dilation caused by the standard compiler-based instrumentation of Scalasca in a certain class of user codes. Specifically, we found that the overhead of measuring codes with a large number of small function calls, such as C++ operators was overwhelming. The *DROPS* C++ application — which we will introduce later in more detail — ran more than 90 times slower with our measurement system, and even after applying aggressive filtering to reduce the overhead, it was still more than three times slower than the uninstrumented version. To enable our users to take meaningful measurements of this class of applications, we had to come up with a new measurement method. Sampling is the well-known alternative to direct instrumentation. The performance analyst has more direct control of the measurement overhead when using sampling, so it was a reasonable solution for our measurement dilation problem. However, sampling only provides statistical data, and does not provide detailed communication metrics, such as number of messages sent or bytes transferred. As having exact and full information

about communication behavior is one of the main goals of taking a Scalasca measurement, and also imperative for our compression algorithm to function properly, a hybrid solution was developed. The new measurement technique uses direct instrumentation for measuring communication events and sampling for the rest of the execution. While integrating data coming from the two kinds of sources simultaneously proved to be challenging, after overcoming some initial problems we were able to create a powerful new hybrid measurement technique, integrating the best capabilities of both sampling and direct instrumentation.

Moreover, we adapted the compression algorithm to the new challenges presented by the hybrid measurement technique. In these measurements, only the data about the communication part of the execution is guaranteed to be exact, the rest is only statistical. For this reason, the comparison rules in the call-tree structure comparison methods of the algorithm had to be relaxed, to accommodate the new kind of data. With the synthesis of these new measurement techniques, a new, significantly more powerful measurement system has been developed, combining robustness, low overhead, and a range of opportunities to gain new insight into the performance behavior of present day HPC applications.

The rest of the document is structured as follows: Chapter 2 introduces a study of a representative set of MPI applications using existing Scalasca capabilities, providing the foundations for our later evaluations. This study is followed by the detailed description and evaluation of our main contributions:

- The implementation and evaluation of *time-series profiling* based on *iteration-instrumentation* in Chapter 3, finding a wide range of non-trivial patterns in the time-dependent behavior of our test applications.
- The design of a novel on-line compression algorithm developed to reduce the memory footprint of time-series profiles is described and analyzed in Chapter 4.
- A hybrid measurement method combining the advantages of both sampling and direct instrumentation is discussed in Chapter 5. It achieves low-overhead measurements while still collecting enough information for our compression algorithm to function properly.
- An implementation that integrates the on-line compression algorithm with the hybrid sampling-based profiling method, with specific adjustments in the compression algorithm to fit the new, less deterministic input data is evaluated in Chapter 6.

Finally, Chapter 7 reviews related work not covered in the introduction and Chapter 8 presents the conclusion and future work.

Table 1.1: *Summary metrics collected and derived by Scalasca.*

Category	Metric
Time	Total
	Measurement overhead
	Execution
	MPI
	MPI init/exit
	MPI synchronization
	MPI collective synchronization
	MPI one-sided synchronization
	MPI one-sided active-target synchronization
	MPI one-sided passive-target synchronization
	MPI communication
	MPI point-to-point communication
	MPI collective communication
	MPI one-sided communication
	MPI file I/O
	MPI collective file I/O
	OpenMP
	OMP flush
	OMP thread management ...
	OMP thread synchronization ...
	Idle threads
	Limited parallelism
	Computational imbalance ...
Counts	Call path visits
	MPI synchronizations
	MPI point-to-point synchronizations
	MPI point-to-point send synchronizations
	MPI point-to-point receive synchronizations
	MPI collective synchronizations
	MPI one-sided synchronizations
	MPI one-sided fences
	MPI one-sided GATS epochs
	MPI one-sided access epochs
	MPI one-sided exposure epochs
	MPI one-sided locks
	MPI one-sided pairwise synchronizations
	MPI communications
	MPI point-to-point communications
	MPI point-to-point send communications
	MPI point-to-point receive communications
	MPI collective communications
	MPI collective communications as source
	MPI collective communications as destination
	MPI collective exchange communications
	MPI one-sided communications
	MPI one-sided puts
	MPI one-sided gets
	MPI bytes transferred
	MPI point-to-point bytes transferred
	MPI point-to-point bytes sent
	MPI point-to-point bytes received
	MPI collective bytes transferred
	MPI collective bytes incoming
	MPI collective bytes outgoing
	MPI one-sided bytes transferred
	MPI one-sided bytes sent
	MPI one-sided bytes received
	MPI file operations
	Individual MPI file operations
	Individual MPI file read operations
	Individual MPI file write operations
	Collective MPI file operations
	Collective MPI file read operations
	Collective MPI file write operations
	Hardware counters (optional)

Chapter 2

Experimental Setup

2.1 Introduction

This chapter gives an overview of the different application codes used throughout the dissertation for testing and evaluating the newly introduced techniques. An extended, more comprehensive set of initial measurement results is provided in Appendix A for reference, using standard capabilities of the Scalasca measurement system.

While Scalasca has two main modes of operation, runtime summarization (aka profiling) and trace collection and subsequent automatic trace analysis, this work expands upon the more basic summarization mode, so no trace measurement results are shown here. This does not mean that the techniques introduced here are generally incompatible with tracing. In Chapter 3 we show how iteration instrumentation is used in tracing measurements.

Furthermore, iteration instrumentation can be used in our implementation to direct tracing to a given subset of the iterations (selective tracing). The intended use case for this kind of selective tracing is that the user identifies the most relevant iterations using a profile measurement combined with iteration instrumentation, and then collects a trace of only those selected iterations, thereby decreasing the measurement's memory footprint.

Based on past experience, we can safely assume that the largest computers currently on Earth will be lagging behind the leaders in two years and history in 4-5 years. It is therefore no wonder that a range of machines was used in the past years to evaluate our techniques, including an IBM Power 4 and a Power 6 system both called JUMP, the Blue Gene/L system JUBL, the Blue Gene/P system JUGENE and several x86_64-based systems, most prominently JUROPA. See Table 2.1 for details. All of the listed machines were main production systems of the Jülich Supercomputing Centre (JSC) at some point. For clarity and better comparability, we present all our results on the JUROPA and JUGENE systems, which are the main workhorses of the Jülich Supercomputing Centre at the time of this writing.

The results presented here cover the SPEC MPI 2007 2.0 benchmark suite [83, 66, 64, 65] with its 18 benchmark codes as well as the *DROPS* computational fluid dynamics code [37] developed at RWTH Aachen and the *PEPC* coulomb-solver [34] developed at Jülich Supercomputing Centre. The *PEPC* code is used as a challenging real-life example in Section 3.3, while the *DROPS* code is used in a similar role in Section 5.3.

Benchmark suites are commonly used for parallel performance tools studies, such as the evaluation of the *Vampir* trace collection and visualization toolset with the 13 applications

Table 2.1: *Characteristics of the JSC computer systems used.*

	JUMP 4	JUMP 6	JUBL	JUGENE	JUROPA/HPC-FF
Vendor	IBM	IBM	IBM	IBM	Sun/Bull
Type	SP2 p690+	p6-575	Blue Gene/L	Blue Gene/P	Constellation
Processors	Power 4+	Power 6	PowerPC 440	PowerPC 450	Xeon X5570
” manufact.	IBM	IBM	IBM	IBM	Intel
” frequency	1700MHz	4700MHz	700MHz	850MHz	2930MHz
” bitness	32/64	32/64	32	32	64
Racks	41	1	8	72	24
Nodes/rack	N/A	14	1 024	1 024	137
Procs/node	16	16	1	1	2
Cores/proc	2	2	2	4	4
Cores	1 312	448	16 384	294 912	26 304
Mem./node	128GB	128GB	0.5GB	2GB	24GB
Mem./core	4GB	2GB	0.25GB	0.5GB	3GB
Agg. memory	5.2TB	1.8TB	4.1TB	144TB	79TB
Filesystem	GPFS	GPFS	GPFS	GPFS	Lustre
Op. system	AIX5.3	AIX5.3	CNK & SLES9	CNK & SLES10	SLES11
Compilers	XLC8,XLF10	XLC10, XLF12	XLC8, XLF10	XLC9, XLF11	Intel 11
MPI	POE 4.2	POE 5.1	BG-MPICH1	BG-MPICH2	ParaStation 5.0
Switch	HPS	N/A	Blue Gene/L	Blue Gene/P	Sun Data Center
Networks	N/A	10Gbps Eth., Infiniband, 1Gbps Eth.	3D torus, Global tree / coll. netw., barrier netw., 1Gbps Eth.	3D torus, Global tree / coll. netw., barrier netw., 10Gbps Eth.	Fat Tree Infiniband QDR
LinPack R_{peak}	8.9TF	8.4TF	45.9TF	1002.7TF	308.3TF
LinPack R_{max}	5.6TF	N/A	37.3TF	825.5TF	274.8TF
Top500 first	#21	N/A	#8	#3	#10
Top500 curr.	N/A	N/A	N/A	#9	#23
Prod. start	Feb-04	May-08	Jul-05	Feb-08	Aug-09
Prod. end	Jul-08	Jan-10	May-08	N/A	N/A

of the SPEC MPI benchmark suite and the *ompP* profiler with the 11 applications of the SPEC OpenMP benchmark suite [22], or the evaluation of *ScalaTrace* using the NAS Parallel Benchmarks.

While the SPEC MPI benchmark codes are derived from real-world applications, we also present some actual real-world application studies where we were able to provide new insight into the performance characteristics of important applications using the techniques introduced in this work, contributing to substantial improvement in the performance of the *PEPC* code, and to the first high-quality, low-overhead performance measurements of the *DROPS* code, which a number of tools including the released version of Scalasca previously were unable to provide [48].

2.2 Experiment configuration

Version 1.0 of the SPEC MPI2007 benchmark suite [83, 66] was released in June 2007 to provide a standard set of MPI-based HPC application kernels for comparing the performance

of parallel/distributed systems' hardware, operating system, MPI execution environment and compilers. This initial release includes 13 applications and a medium-sized 'mref' reference dataset (MPIm2007) for benchmarking runs requiring up to 2GB of memory per process and configurable for up to 512 processes. As the benchmarks are configured for strong scaling, that is, solving the same problem size at different process counts, their running times became very short at larger scales, limiting the value of studying the performance at those scales.

This problem was partially remedied with the release of version 2.0 in February 2010 and the introduction of the large-sized 'lref' reference dataset, which scales up to 2048 processes, provides much better scalability and more reasonable run times at larger scales. Version 2.0 introduced one new application (*125.RAxML*) and updated versions of four applications, bringing the total count to 18 benchmark codes.

Table 2.2 summarizes the 18 benchmark codes of the MPI2007 suite as well as the *PEPC* and *DROPS* codes, showing that they derive from a wide variety of subject areas and are implemented using a representative cross-section of programming languages (C/C++/Fortran, often combined). As a convention, the SPEC MPI applications that only have an 'mref' medium sized dataset are shown with a grey background. From the MPI usage breakdown in the table, it can be seen that a variety of MPI functions are used at many locations ('sites') in the source code, however, performance analysis can concentrate on the smaller number of communication and synchronization functions (shown as c&s/used 'funcs') and the distinct program call-paths on which they are actually executed during benchmark runs ('paths').

2.2.1 Initial measurements

We compile the SPEC MPI codes with the Intel 11.1 compilers and run them with ParaStation MPI 5.0 on the JUROPA system. The *DROPS* code is also evaluated using the same machine configuration, while we make an exception with the *PEPC* code and evaluate it using the IBM Blue Gene/P system JUGENE, the system where it is regularly used in production runs.

There were some compile/link/run-time issues with some of the applications using the Intel 11.1 compilers. In these cases we resolved the issues by falling back to the Intel 11.0 compilers in some cases, or progressively removing aggressive optimizations in others, until a viable application executable was produced. This is a typical procedure for SPEC MPI benchmarking, though we were not aiming to strictly follow SPEC benchmarking rules. Full optimization of the code and run-time environment were neither essential nor particularly desirable for our purposes, as the study is focused on 'typical' application performance in a representative HPC environment rather than on benchmarking.

The scalability of the applications in our test suite is shown in Figure A.1, on page 143. The programs generally exhibit good scalability for fixed problem sizes: 1024-process runs complete in a few minutes in most cases, although the scalability of *142.dmilc* and *128.GAPgeofem* is poor after 256 processes. We prefer longer measurement runs since run-to-run variation of several seconds (e.g., due to use of the shared file system) complicates performance comparisons. Thus, we focus on the 256-process executions.

Table 2.2: *Test applications' coding and subject area. Application name, available test sizes, language used, code line count, MPI function count (communication&synchronization/all), MPI call-site count (all), MPI call-path count (all), subject area. Shaded rows indicate applications from the old version of the SPEC MPI benchmark suite, which are not analyzed in detail in this thesis.*

	Test size	Program		MPI			
Application code	available	language	LOC	funcs	sites	paths	Application subject area
104.milc	**mref**	C	17 988	6/11	51	119	Lattice quantum chromodynamics
107.leslie3d	**mref**	F77,F90	10 503	6/10	34	11	Combustion dynamics
113.GemsFDTD	**mref**	F90	21 867	6/12	234	28	Computational electrodynamics
115.fds4	**mref**	C,F90	44 524	6/10	239	13	Computational fluid dynamics
121.pop2	mref, **lref**	F90	104 150	6/13	152	228	Oceanography
122.tachyon	mref, **lref**	C	15 539	8/13	17	18	Computer graphics: ray tracing
125.RAxML	**lref**	C	36 192	2/6	209	353	DNA matching
126.lammps	mref, **lref**	C++	56 731	8/16	626	78	Molecular dynamics
127.wrf2	**mref**	C,F90	208 256	5/10	198	82	Numerical weather prediction
128.GAPgeofem	mref, **lref**	C,F90	30 964	5/10	55	17	Geophysics finite-element methods
129.tera_tf	mref, **lref**	F90	6 509	7/11	40	19	Eulerian hydrodynamics
130.socorro	**mref**	C,F90	91 769	8/13	155	156	Quantum chemistry
132.zeusmp2	mref, **lref**	C,F90	44 297	7/15	607	91	Astrophysical hydrodynamics
137.lu	mref, **lref**	F90	5 823	7/12	59	29	Linear algebra SSOR
142.dmilc	**lref**	C	17 993	6/10	51	117	Lattice quantum chromodynamics
143.dleslie	**lref**	F77,F90	4 347	6/10	34	12	Combustion dynamics
145.lGemsFDTD	**lref**	F90	21 431	8/14	217	29	Computational electrodynamics
147.l2wrf2	**lref**	C,F90	240 546	8/16	221	110	Numerical weather prediction
DROPS		C++	159 932	15/24	495	475	Computational fluid dynamics
PEPC		C,F90	22 186	11/16	421	67	Plasma physics and astrophysics

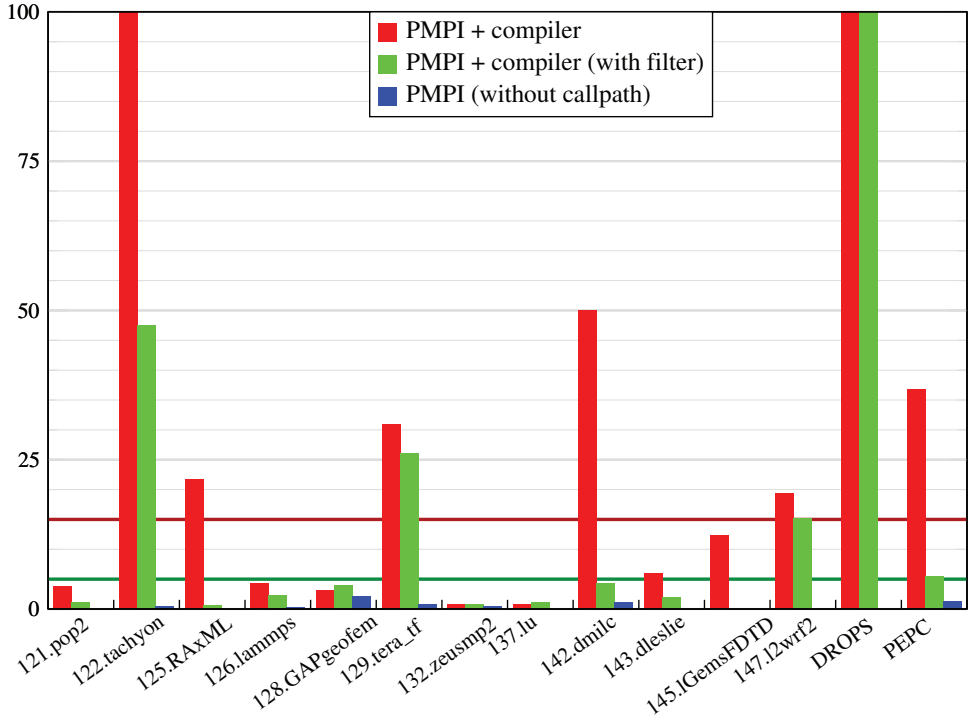


Figure 2.1: *Measurement overhead percentage for executions with compiler instrumentation of all user-level source routines (also employing filtering) with call-path profiling, and basic PMPI-only routine profiling.*

With this processor count, we measure each code three times and observe under 5% run-to-run variation in most cases. However, some executions took considerably longer than the average. To reduce the impact of this variability, we use the fastest of the three runs as the reference time for an uninstrumented execution, which we compare to the fastest runs with Scalasca to assess measurement overheads.

The measurements employ the Scalasca measurement library in its default runtime summarization mode, which produces an analysis report that integrates call-path profiles from each process. The measurements include a full complement of wrappers for MPI routines, enabling capture of `MPI_Init` and `MPI_Finalize` and all other MPI events according to its runtime configuration.

If the user chooses to disable compiler-based instrumentation, the Scalasca measurement library is simply linked into the executable, including all MPI wrappers. This provides the most basic Scalasca measurement possible, consisting only of a (flat) profile of MPI routines. We expect this measurement to have the lowest overhead, providing comprehensive MPI communication and synchronization statistics, albeit without call-path contexts. The rightmost (blue) set of bars in Figure 2.1 shows that, while *143.dleslie* execution took 3.5% longer than the uninstrumented reference, the other applications (including *DROPS*) were dilated

less than 1%, confirming the low overhead of basic PMPI profiling with Scalasca. While unnecessary for these applications, we could disable groups of MPI events if required to reduce measurement overhead further.

Figure 2.1 gives an overview of Scalasca measurement dilation in the different basic measurement modes. Only those SPEC MPI applications with 'lref'-sized datasets are included here, as 'mref'-sized datasets are generally too small to be relevant at the 256-process scale we use for evaluation. The red bar shows dilation percentage using the PMPI profiling interface combined with compiler instrumentation for user function entry and exit events.

Given that we consider any measurement with dilation above 15% (red line) to be seriously perturbed for our current purposes, overhead ratios shown in this graph for around half of the applications are unacceptable. High measurement overheads are a common problem among parallel performance systems [48], and care must be taken when applying the Scalasca (or any other) measurement system to measure the performance characteristics of any application. Just taking a measurement without any precautions and naively assuming that it is representative of the actual application execution behavior is dangerous. This is perhaps the most important rule of performance analysis tools, and users must be trained to be aware of this fact. A wide variety of techniques have been developed to circumvent these problems and bring the measurement dilation down to acceptable levels (preferably under the green line at 5%) in as many cases as possible. In some cases quite simple methods suffice, while in others more complex or manual solutions have to be employed to fine-tune the measurement [27, 67].

A simple way of reducing overhead is to turn off compiler-based instrumentation entirely and rely solely on PMPI wrappers. As the blue bars in the figure show, it reduces overhead to essentially negligible levels in nearly all cases. Turning off all compiler-based instrumentation is certainly an effective way of reducing overhead, but it causes significant losses in the amount of information our measurement is able to extract from an execution. While the counts and timings of MPI operations are collected, without instrumentation of the user source routines making the MPI calls, it is not possible to resolve which calls from which part of the application are most critical to overall performance. In some cases this amount of information is enough to get an idea of the application's performance characteristics. In other cases, we need more, and we need to use more elaborate methods.

Excessive measurement dilation is usually caused by small, generally not very important functions which are called extremely frequently. As the measurement system is called and records every entry and exit to all of these functions each time they are called, these small functions can easily have devastating effects on measurement overhead. A semi-automatic method of dealing with these functions is run-time measurement filtering. Based on the initial measurement we use a post-processing tool that generates a list of the functions in the measured application in decreasing order of *visit counts*. Furthermore, it also lists the relative time spent in each function. The ones with high visit count and low time values are the best candidates for filtering. The tool also marks every function as belonging to different categories, based on what call paths they appear on [73, 75]. We refer to these call-path categories often in the rest of the document:

- *MPI call paths* are calls to MPI functions.
- *COM call paths* call MPI call paths directly or indirectly.

- *USR call paths* do not call MPI or COM call paths, neither directly nor indirectly.

Functions appearing on COM or MPI call paths should not be filtered, as we generally want to be able to differentiate between calls to the same MPI function from different call paths. USR functions — especially the ones where very little time was spent — can safely be filtered as they usually do not have an immediate impact on our understanding of the communication characteristics of the application. USR functions are useful for understanding the performance characteristics of local computations and probable causes of workload imbalances, but small, frequently called functions would introduce too much dilation and contribute too little to the understanding to justify keeping them.

By ignoring entry and exit events of these functions, overhead can be substantially reduced while not losing too much valuable information. The time spent in the filtered functions will not be lost to our measurement, it will show up in their respective parent call paths, as if they were inlined into their calling functions. The green bars in the figure show that (runtime) filtering is a good compromise between the two extremes, bringing the dilation to acceptable levels in nearly all cases. *122.tachyon* and *DROPS* still show very significant overhead. In these cases instrumented events appear so frequently that simply making the run-time decision to ignore them takes too much time. In these cases, more labor-intensive solutions are available, like selective instrumentation (aka static filtering). In Chapter 5 we will look at some alternative methods of reducing measurement overhead while still collecting detailed performance measurements.

Table 2.3 shows application execution characteristics determined from the 256-way Scalasca runtime summarization experiments. Application programs are seen to typically consist of hundreds to thousands of global timesteps or solver iterations, with the farming-based *122.tachyon* being an exception. Although *130.socorro* only does 20 iterations, it has by far the most complex call-tree of all SPEC MPI applications. The deepest call tree is that of *125.RAxML*, which has MPI calls as deep as 36 levels from the root. The largest and most complex call tree is that of *DROPS*, which is also the application that uses the widest selection of MPI calls. *DROPS* was specifically selected for this purpose, to evaluate our methods on an extraordinarily complex C++ application.

The memory usage data of the applications shows that most applications have relatively modest memory requirements at the 256 process scale, in the 50-300MB range, with some as high as 600-1000MB per process. Also, the average and maximum memory consumption of the processes roughly match in most cases, indicating that all processes are doing similar work, with the exceptions *126.lammps*, *147.l2wrf2* and *DROPS*, which show significant differences. In most cases this is because the master process does some additional processing which requires more memory.

Although this work is not about the tracing capabilities of Scalasca, trace buffer usage is a good measure of the effectiveness of a filter, so we included trace buffer-related metrics in this table. The trace buffer usage of a measurement is roughly proportional to the number of events recorded during measurement, a good indicator of the event frequency, which in turn is strongly related to the measurement dilation level. Where the trace buffer usage is low, the measurement dilation can also be expected to be low.

Table 2.3: *256-way test application execution characteristics.*

	Program execution				Memory (MB)		Trace buffer content (MB)				Filter
Application code	time	steps	depth	call paths	avg.	max.	total	MPI	filter	residue	funcs
104.milc	35	8+243	8/8	535/543	78	81	9 648	118.2	9 410	1	40/255
107.leslie3d	110	2 000	4/4	47/47	81	83	623	13.9	605	8	7/38
113.GemsFDTD	301	1 000	5/9	337/373	108	124	136 064	4.9	136 019	8	14/287
115.fds4	56	2 363	3/12	251/256	347	348	636	9.7	623	3	15/326
121.pop2	461	440	7/7	620/621	934	962	1 828	77.0	1 613	132	9/1 443
122.tachyon	478	-	8/12	344/346	637	645	259 244	0.5	259 244	0	55/413
125.RAxML	630	21 268	36/36	2 472/2 515	298	309	20 613	2.6	20 607	9	8/403
126.lammps	574	300	7/8	473/473	66	94	1 605	0.9	1 587	17	8/1 581
127.wrf2	113	1 375	18/159	5 444/5 460	145	169	1 252	30.3	1 179	41	70/3 582
128.GAPgeofem	575	2 501	5/5	60/60	617	679	2 143	578.3	1 593	2	2/66
129.tera_tf	281	190	6/6	100/102	239	239	1 591	3.2	1 585	2	9/83
130.socorro	103	20	19/23	10 360/10 362	112	113	1 991	14.9	1 957	20	39/2 067
132.zeusmp2	247	200	6/6	177/185	236	236	5	3.8	0	0	1/198
137.lu	269	180	5/5	66/71	43	45	210	26.8	169	3	1/47
142.dmilc	186	8+241	8/8	534/540	308	309	36 868	463.1	36 399	4	41/255
143.dleslie	251	15 054	4/4	46/46	55	55	2 271	103.4	2 162	36	7/36
145.lGemsFDTD	326	1 500	5/9	328/368	263	305	6 139	6.2	6 076	13	13/280
147.l2wrf2	867	720	17/17	1 763/1 790	339	538	4 787	36.6	4 565	174	31/2 380
DROPS	5841	250	20/28	19 246/18 676	68	203	853 252	1356.6	784 313	67 808	3 161/3 687
PEPC	12836	1 300	6/6	172/172	238	254	21 325	263.2	21 105	0	13/57

As the table shows, in many cases the trace buffer would require gigabytes or even hundreds of gigabytes of memory without filtering, which can be reduced to much more realistic levels (in many cases to just a few megabytes) by filtering out a handful of functions, usually less than a dozen. In the case of *DROPS*, we filtered out every user function not on a call path to MPI calls, but we still encountered very significant overhead.

Note that in the max individual/total unique call paths column of Table 2.3 there are contradicting numbers shown for *DROPS*, the individual number being higher than the maximum number. This is due to bugs in the Intel C++ compiler instrumentation of routines. These bugs cause multiple routines to have incomplete or missing names, and after unification of call trees these are no longer distinguished. While the presence of such errors is a nuisance, by automatically marking the occurrences of these anomalies in the call tree we were able to reduce the impact of these bugs to a negligible level.

Chapter 3

Performance Dynamics

In this chapter we build on the concept of phase-based performance characterization. There are several, slightly different definitions for *phases* in the literature, which we discuss later in Section 7.1. In our definition, the execution of a program can be naturally divided into phases, which form the building blocks of its performance behavior. Since phases have different execution characteristics and may react differently to external stimuli such as the change of the execution configuration or of the input problem, it seems reasonable to analyze their performance behavior independently instead of looking at the execution only as a whole. While phases provide a general concept to represent arbitrary logical and runtime aspects of the computation, our approach concentrates only on major timestep loop iterations to allow comparisons between execution intervals that occur on the same logical level. With this method we build on the fact that applications tend to be iterative in nature, repeating roughly the same sequence of operations until some stoppage criterion is fulfilled. This can be convergence to a final result or reaching the end of the simulated time interval (in time-stepping loops). Moreover, applications often have a single main iterative loop, where the majority of the work is done and most of the execution time is spent. Later we will show an example where this latter assumption does not hold, the *142.dmilc* application, where there are five distinct phases of execution, each having their own main loop, and significant amounts of execution time spent in each of those phases. This special case, while not impossible to handle correctly, falls beyond the scope of this thesis.

As we have seen in Section 1.5, going beyond standard summaries of a complete execution, and comparing performance phenomena occurring in separate iterations has great potential for providing new and interesting insight into application behavior. In particular, performance behavior may vary between individual iterations, for example, due to periodically re-occurring extra activities [51] or when the state of the computation adjusts to new conditions in so-called adaptive codes [80]. Motivated by such examples, a source-code instrumentation API was implemented that provides a straightforward way for collecting data from iterations separately. An example of the code necessary for marking the main iterative loop of an application is shown in Listing 3.1. The macro and variable names used in the instrumentation all start with "epik_" or "EPIK_", the name of the measurement system. In the most straightforward case iteration instrumentation could be as simple as marking the beginning and the end of the body of the main loop (see the lines referring to `epik_phase_iteration`). In this figure, we introduced some additional phases to separate the initialization and finalization part of the execution from the core part, but the necessary source-code instrumentation is still less than

Listing 3.1: *Typical example of manual iteration instrumentation in C source code.*

```
1  // include user instrumentation header
2  #include "epik_user.h"
3
4  int main(int argc, char *argv[]) {
5    int epik_phase_setup, epik_phase_core;
6    int epik_phase_finalize, epik_phase_iteration;
7
8    // store the phase id in the variable and provide name
9    EPIK_PHASE_REGISTER(epik_phase_setup, "SETUP");
10   EPIK_PHASE_REGISTER(epik_phase_core, "CORE");
11   EPIK_PHASE_REGISTER(epik_phase_finalize, "FINALIZE");
12   EPIK_PHASE_REGISTER(epik_phase_iteration, "ITERATION");
13   EPIK_PHASE_START(epik_phase_setup);
14
15   // setup and initialization routines
16   ...;
17   EPIK_PHASE_END(epik_phase_setup);
18   EPIK_PHASE_START(epik_phase_core);
19
20   // main loop
21   for(i=0; i<100; i++) {
22     EPIK_PHASE_START(epik_phase_iteration);
23     // main loop contents
24     ...;
25     EPIK_PHASE_END(epik_phase_iteration);
26   }
27   EPIK_PHASE_END(epik_phase_core);
28   EPIK_PHASE_START(epik_phase_finalize);
29
30   // finalization routines
31   ...;
32   EPIK_PHASE_END(epik_phase_finalize);
33 }
```

20 lines. Compared to the size of the codes we are typically analyzing, this is negligible. Also, in optimal cases we are working together with the application developers who can easily point out the main loop of the application in a matter of minutes, but even when not having access to application developers our experience shows that finding the main loop of an application based on the visit counts (number of calls on each call path) seen in a standard Scalasca measurement is quite straightforward and usually does not take more than an hour.

To clarify our usage of these two important terms in the rest of the document:

- *Phase* refers to the execution of a specific part of the application source code, whose instances constitute a contiguous interval in time during execution.
- *Iteration* refers to a single execution instance of a repetitively executed phase. In the code it is typically the body of a loop structure.

Note that it may be expected that marked phases and iterations appear the same number of times at the same part of the call tree on every process. This is an intuitive expectation from the user's point of view, but this is neither required nor guaranteed by our infrastructure. In most cases it is true, but this only depends on the application being measured.

It is also interesting to note that unlike some other tools [58], we don't differentiate between phases and iterations in the marker API, we use a single kind of marker for both cases. By default, we handle everything in the same way, so if they are executed repetitively, all instances will be collected separately. If the user needs fine-grained control over collecting every instance of certain marked regions separately and aggregating others, this capability is provided through a runtime configuration file. The advantage of this approach over using different kinds of markers in the API is that there is no need to recompile and relink the application in order to change the measurement configuration. We gain greater flexibility at the cost of negligible runtime overhead.

For evaluation purposes, we looked at each application of our test suite, and for those with identifiable repetitive phases, corresponding to global timesteps or solver iterations, we inserted this additional source-code instrumentation. This kind of instrumentation was only applied to the applications that provide a large reference dataset, as we needed reasonably long run-time at the scale of 256 processes for our evaluation. Also, most of the applications that only had a medium reference dataset were simply older versions of benchmarks that had a large reference dataset, so using both would have been duplicate usage of essentially the same benchmark. Manual iteration instrumentation was possible for all except *122.tachyon* which is based on a task-farming parallelization. In the case of *142.dmilc*, which has a complex structure of nested loops and branches, we instrumented the most significant loop, which still only takes around 20% of the overall execution time. As Table 3.1 shows, the overhead of this additional instrumentation during measurement is in most cases sufficiently low. It is mostly at the same level as the run-to-run variation between executions of the same application.

Here we introduce a formal definition of these *time-series profiling* measurements to ensure clarity in our discussions later on. Let C be the set of call paths reached from within the timestep loop body (those outside can be ignored because they are irrelevant to draw comparisons between iterations), I the set of loop iterations, P the set of application processes,

Table 3.1: *Test applications' execution wall-clock times in seconds and relative execution times at different instrumentation levels.*

		Instrumentation		
Application code	None	Without filtering	With filtering	With filtering & iterations
121.pop2	461	103.7%	101.1%	102.0%
125.RAxML	630	121.8%	100.6%	103.8%
126.lammps	574	104.4%	102.3%	101.9%
128.GAPgeofem	575	103.1%	104.0%	104.2%
129.tera_tf	281	131.0%	126.0%	126.3%
132.zeusmp2	247	100.8%	100.8%	103.2%
137.lu	269	100.7%	101.1%	101.1%
142.dmilc	186	149.5%	104.3%	115.1%
143.dleslie	251	106.0%	102.0%	105.6%
145.lGemsFDTD	326	112.3%	99.7%	99.7%
147.l2wrf2	867	119.4%	115.1%	115.1%
DROPS	5841	9493.7%	202.6%	202.3%
PEPC	12836	136.8%	105.4%	106.1%

and M the directly measured metrics among the ones listed in Table 1.1. Then we can define a *time-series call-path profile* as a mapping

$$t : (c, i, p) \mapsto \vec{m}$$

which maps a call path $c \in C$, an iteration $i \in I$, and a process $p \in P$ onto a vector of metric values $\vec{m} \in M_1 \times \ldots \times M_m$ (e.g., the time and visit count values of thread t in call path c while executing iteration i). Note that the derived metric values are implied in this definition. The process dimension P can be omitted when considering the data of only a single process, which will often be the case, for example, in Chapter 4, as the compression algorithm introduced there is a purely process-local operation.

3.1 Analysis methods

The variety of different analysis techniques made possible by iteration instrumentation is illustrated here through the example of the SPEC MPI application *132.zeusmp2*. The iteration-augmented call-path profiles can be analyzed in several ways. The most common methods are listed below, and illustrated in Figure 3.1.

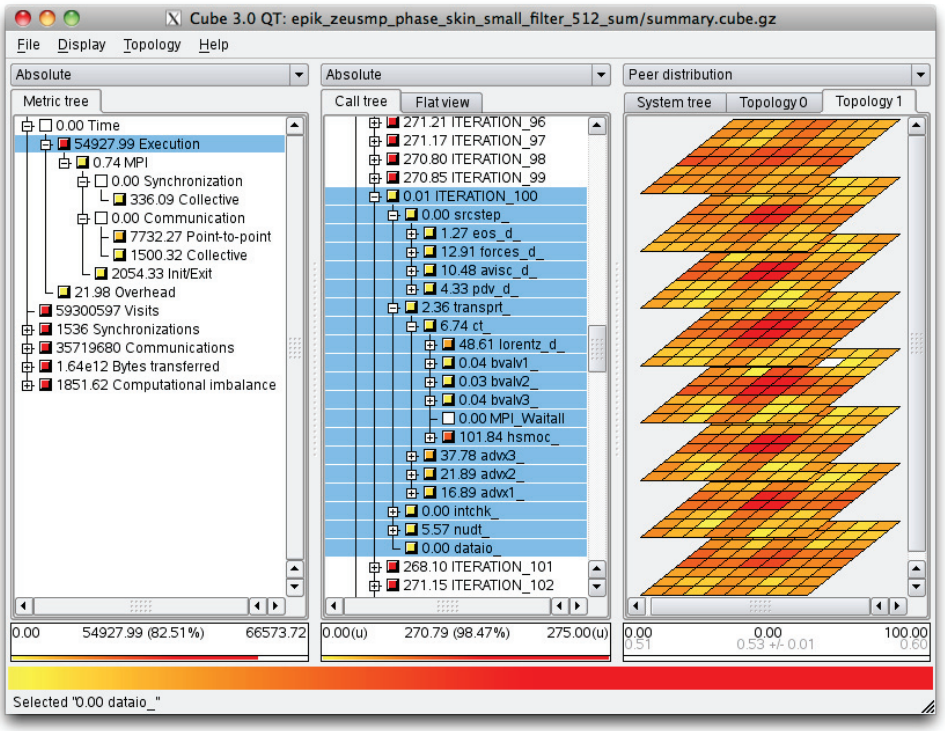
- The Scalasca analysis report explorer, a multidimensional tree browser [25] that allows a user to interactively explore the entire performance space spanned by mapping t including the derived metrics and, in particular, to examine the performance behavior of individual call paths (Figure 3.1(a)).
- 2-dimensional iteration graphs, which for each iteration (x-axis) show the maximum, median, and minimum values of a given metric (y-axis) calculated from all processes (Figure 3.1(b)).

- Value maps, which display the value of a given metric color-encoded as a rectangular block in a 2-dimensional grid with the x-axis representing iterations and the y-axis representing processes (Figure 3.1(c)). There are several coloring modes available, such as linear gradient between the minimum and maximum values, linear gradient between 0 and the maximum value, or histogram-equalized coloring for maximum contrast.

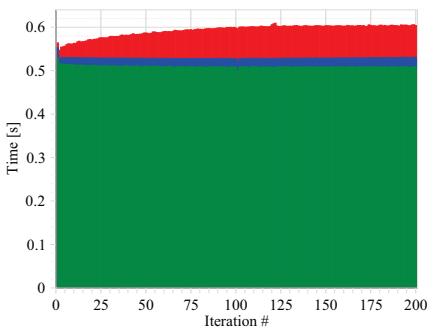
Figure 3.1(a) illustrates the use of the standard Scalasca analysis report explorer GUI with an iteration-instrumented profile measurement. From the metric pane in the left tree the *Exclusive execution time* metric is selected. The term *exclusive* means that we only include the value at this level of the metric hierarchy, as opposed to *inclusive*, where the values of all the sub-metrics would also be included. In this particular case, exclusive execution time means pure computation time, while inclusive execution time means the overall execution time, including computation and all kinds of communication time. In the middle pane the dynamic call tree of the application includes the iterations of the main loop. Each iteration can be analyzed individually, by expanding its call tree and looking at the performance characteristics of its individual call paths.

An example of this is iteration 100, shown here with an expanded call tree. In this case we found that most *Exclusive execution time* is spent in the function `hsmoc_`, 101.75 of the overall 271 seconds aggregated from all 512 processes. However, iteration 100 is selected in the middle pane, which means that the overall *Exclusive execution time* of that iteration is displayed for each process in the right pane. As *132.zeusmp2* is one of the applications that uses `MPI_Cart_create` to set up a virtual process topology, Scalasca was able to automatically capture this call and present us with the virtual topology view here in the GUI. The color of each square indicates the metric value of the corresponding process in the three-dimensional, 8x8x8 topology. The color scale is shown at the bottom of the window, the yellow end of the spectrum representing low values and the red end representing high values. The highest computational load is found in a spherical region in the middle of the topology, and the values become lower as the distance from the middle grows. *132.zeusmp2* simulates a blast, which takes place in the center region. Thus, it is safe to assume that the processes on the edges are just taking care of the boundary conditions while the main problem is actually focused in the middle, with correspondingly higher computational workload. This is clearly an important source of workload imbalance, which is likely to deteriorate performance at scale, as the processes with less workload will always finish earlier and have to wait for the others at synchronizing communication calls. More detailed analysis of the workload imbalance in *132.zeusmp2* is available in [10].

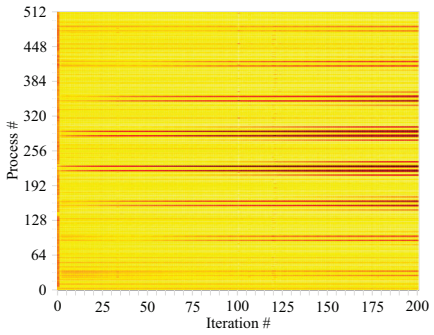
Figure 3.1(b) should be interpreted as follows: The horizontal axis identifies the iterations, whereas the vertical axis corresponds to the metric values in those iterations. The metric used here is *Exclusive execution time*. Green, blue and red bars show the minimum, average and maximum values, respectively across all processes in the given iteration. For *132.zeusmp2* the minimum and the average values are both constant over time, the average being 0.53 seconds per iteration for all but the first iteration that takes slightly longer. This indicates that most processes are likely to have a constant workload, which is generally a good sign. In contrast, the red part of the graph shows that one or more processes take progressively more time, growing from 0.55s initially to 0.61s at iteration 200.



(a) Scalasca analysis report explorer with *Exclusive execution time* metric selected (left pane). The call tree of iteration 100 is opened up for inspection and selected in the call-tree in the middle pane, with its values displayed in the application's 8x8x8 virtual process topology on the right.



(b) *Exclusive execution time* iteration graph with maximum (red), median (blue), and minimum (green) values.



(c) *Exclusive execution time* value map with the red end of the color spectrum indicating higher values (i.e., 0.61 seconds).

Figure 3.1: *Different ways of analyzing a 512-process iteration-instrumented 132.zeusmp2 experiment.*

We get a better understanding of the situation by examining Figure 3.1(c). This color-coded value map is based on the same data as Figure 3.1(b), it is just a different view. The horizontal axis still identifies the iteration number, but the vertical axis corresponds to the process rank. For a given iteration-rank pair, the corresponding color on the chart shows the metric value using a linear scale from the minimum to the maximum value of any process in any of the iterations i.e. 0.52 to 0.61 seconds. The darker the color, the higher the value. The MPI ranks corresponding to the sphere in the middle of the topology as seen in Figure 3.1(a), are the horizontal dark lines in the chart, whereas the yellow background color corresponds to the ranks closer to the edges. In line with our observations from Figure 3.1(b), the workload on the processes further away from the center is relatively constant, while the progressive growth seen in the maximum (red) part happens on the processes in the central region of the virtual topology, which means that the workload imbalance gets worse over time.

While we have only shown the analysis of a single metric of *132.zeusmp2* here, the power of iteration instrumentation is already clear from this example. We used 512 processes here to get a more insightful, 8x8x8 topology, as opposed to the 8x8x4 used in the 256-process case. However, from here on all analysis of the SPEC MPI applications will use 256 processes.

3.2 Temporal patterns

Figure 3.2 gives an overview of the *Inclusive execution time* metric of the 11 SPEC MPI applications where we instrumented iterations. Some classes of behavior can already be identified from this very coarse overview, such as *129.tera_tf* and 132.zeusmp2 both showing a gradual increase in iteration time, or *126.lammps*, *137.lu*, *142.dmilc*, *143.dleslie* and *147.l2wrf2* all showing a relatively flat baseline with periodic peaks at the same time on all processes. *121.pop2* seems similar but not quite as flat in the baseline. After more detailed analysis we see that despite the very similar behavior seen at this level of examination the underlying performance characteristics vary significantly among these applications. Appendix B provides a detailed description of the temporal patterns along with a relevant set of supporting graphs and charts for each application.

3.3 Case study: PEPC

PEPC [34] is a 3-dimensional particle simulation code which employs a hierarchical, parallel tree algorithm implemented using MPI to compute the forces on the particles. The code is presently used for various applications in plasma physics and astrophysics. According to the author of the code, the version of the code we analyzed had potential bottlenecks in the domain decomposition routine, tree construction and tree 'walk', the last of which requires significant point-to-point communication of multipole information between processors, and is thereby sensitive to load imbalance. At the time of our study they did not know what exactly caused the bottleneck, and our study played an important part in their understanding of the problem. Since then a new version of the code has been developed, which overcomes these problems, partially due to the insight gained by our analysis.

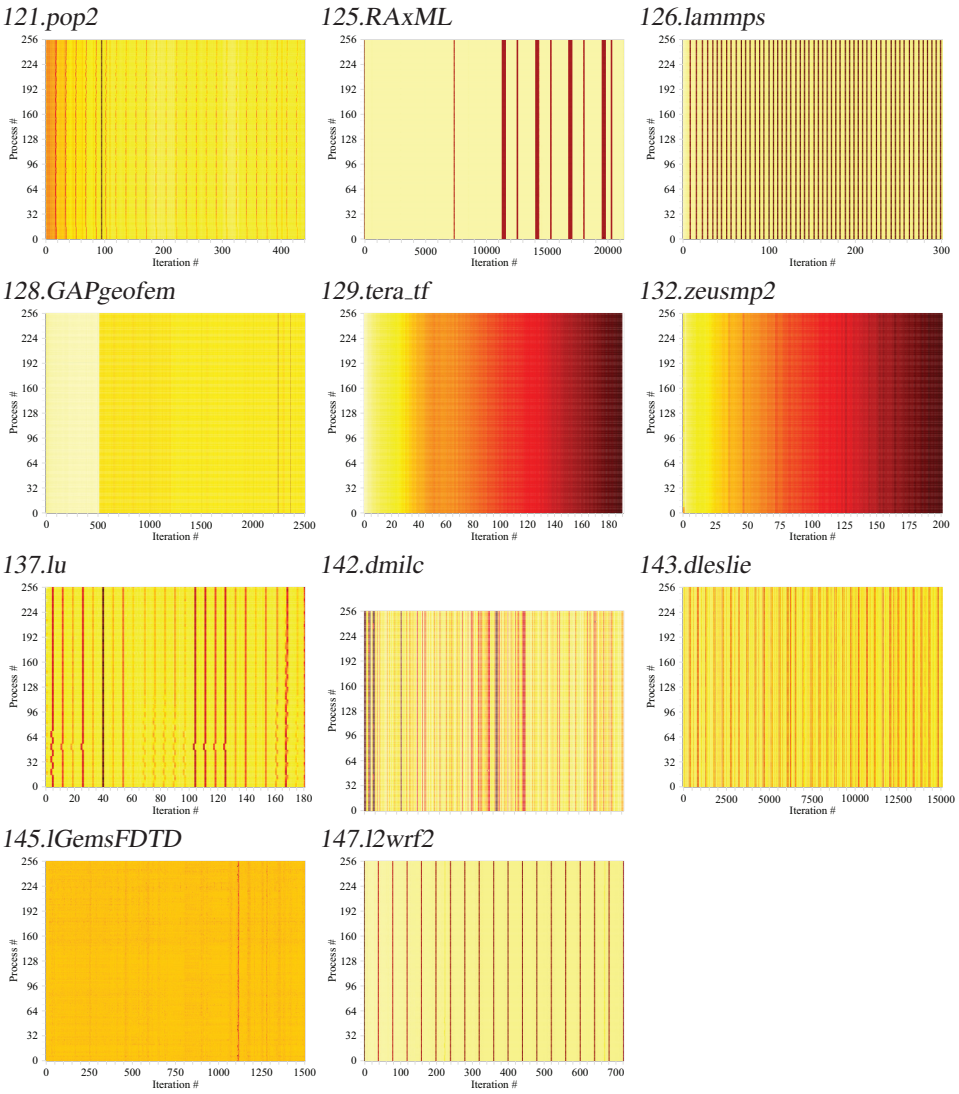


Figure 3.2: *Iteration inclusive execution time maps of the test applications.*

In this study [86], *PEPC* was analyzed using the extended version of Scalasca providing the additional capability of iteration instrumentation. This means that after manually identifying the main time-stepping loop of the application and inserting markers around the loop body, subsequent Scalasca measurements become aware of individual iterations and support the analysis of the time-dependent behavior of the application. Whereas this thesis focuses primarily on runtime summary measurements, for this case study a trace measurement was collected and subsequently analyzed using Scalasca's automated communication-pattern analysis framework. This resulted in additional metrics being collected and incorporated into the analysis report, which require comparison of measurements from different processes. An example is the *Late sender time* associated with an early receive blocking until the corresponding send is initiated, which is a subset of the *MPI point-to-point communication time*. While most of the metrics used in this analysis are available in standard summary measurements as well, we chose to introduce this example using trace measurements as a few of the trace-based metrics were used to gain important insight in the analysis process of this application. After considering examples of such *PEPC* execution analyses and presentations, these are assessed to determine requirements for the potential future integration of the employed iteration-profiling capabilities with Scalasca.

3.3.1 Experimental results

A 1024-way test case, which was run for 1,300 timesteps on the JUGENE system, is used here as an example to show what we have learned about the performance characteristics of the application using the tools provided by the extended Scalasca toolset. This example was chosen as it shows not only interesting patterns of time-varying behavior, but also performance problems, such as serious communication imbalance growing rapidly over time.

Figures 3.3 and 3.4 show an example of the usefulness of having analysis data individually for all the iterations instead of having just the whole program execution. Figure 3.3 shows the case where the iterations are not distinguished and only aggregate metrics are available. When looking at the *MPI point-to-point communication count* metric, a few processes appear as hot-spots in the topology pane, showing that there is some imbalance. This is important to recognize, however, additional insight can be extracted from the extended, iteration-instrumented analysis. When selecting in turn the individual iterations distinguished in Figure 3.4, the execution behavior can be observed evolving over time. During the first iterations, the communication is relatively balanced and there are no extreme hot-spots visible. Over time some hot-spots appear and become increasingly pronounced, and they move at slightly different speeds from one MPI rank to the next. The number of hot-spots diminishes towards the end of the 1,300 steps, however, the severities of the highest ones are rapidly increasing.

Figure 3.5 shows four different views of the same metric, *point-to-point sent message count*, to compare the different kinds of information they provide. The *phase graph* (upper left) gives an overview of the evolution of the values over time. Looking at this graph it is obvious that there is a serious communication imbalance in the application, as the minimum and median values are relatively low and constant throughout the execution, but the maximum value is growing

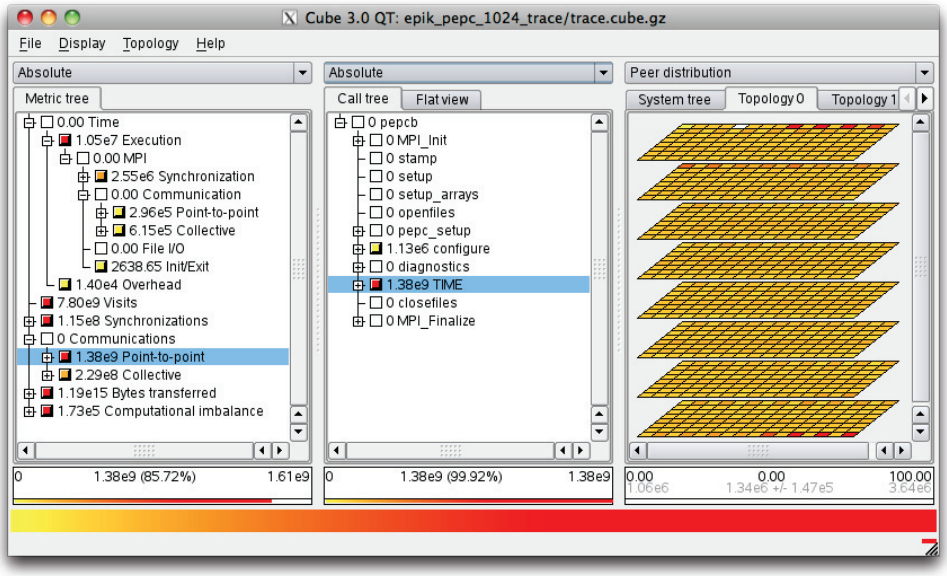


Figure 3.3: *Scalasca analysis-report explorer showing a PEPC trace experiment with significant imbalance in the number of point-to-point communications (selected in left pane). The 1,024 application processes are arranged according to the Blue Gene/P physical network topology (right pane).*

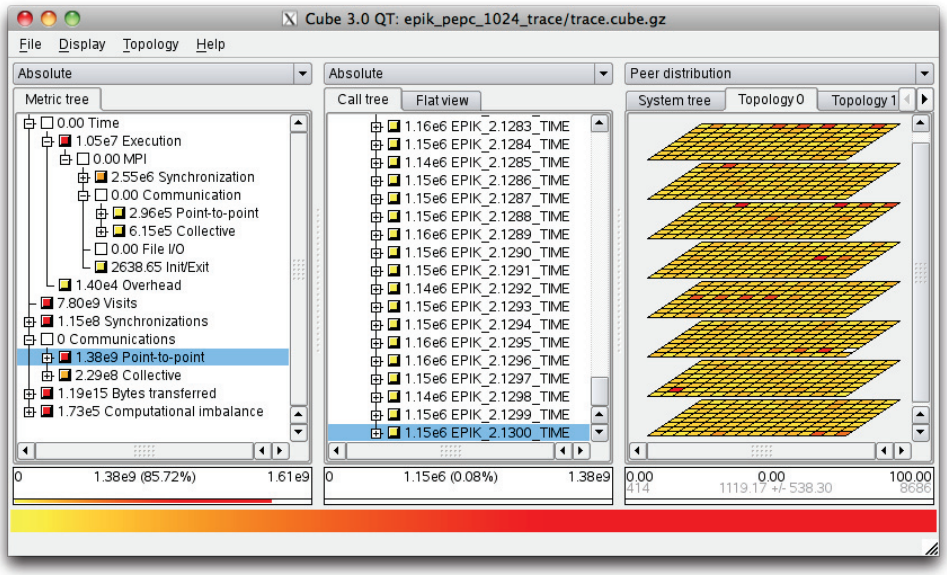


Figure 3.4: *Scalasca analysis-report explorer showing a PEPC trace experiment with timesteps distinguished via iteration instrumentation (center pane).*

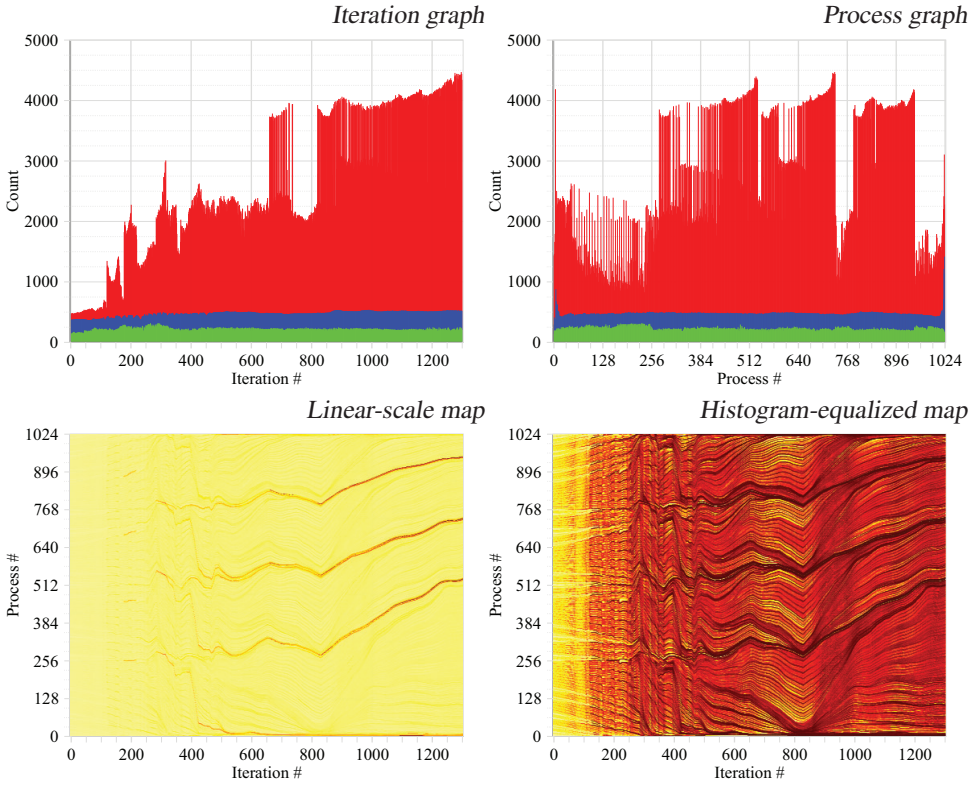


Figure 3.5: *Different analysis presentations of the point-to-point sent message count metric in PEPC.*

rapidly, and it rises many times higher than the minimum or median values. This suggests that within each timestep most of the processes are behaving in a relatively balanced fashion, but that there are a few processes being involved in many more communications.

Which processes are responsible for the imbalance? On the *Process graph* (upper right), the x-axis shows the different process ranks, so the different colors now distinguish the minimum, median and maximum value across all timesteps for the given process rank. It is not obvious which process is responsible for the imbalance, as many processes show very high maximum values. The minimum and median values, however, are consistently low for all processes (except for the first and last few ranks where the median is somewhat higher), which suggests most of the iterations have low values on all processes.

The case is getting increasingly confusing, and we still do not see what is going on here with these high values, but the *linear-scale map* on the lower left makes it much clearer. On a linear-scale map, the x-axis shows the iteration number, the y-axis the process rank, and the values are color-coded from light yellow (for the lowest value) to dark red (for the highest value, here around 5000). The map shows that at the beginning all processes start off relatively balanced, however, after a few hundred timesteps a low number of hot-spots gradually appear whose values get much higher over time than the average. What is interesting about these hot-spots

is that they are not bound to any specific process, but rather they seem to move to neighboring processes in a seemingly coordinated manner. As they migrate, some hot-spots appear to merge, so after around 700 timesteps only five hot-spots remain (the first and last MPI ranks and three others in between), each consisting of a few processes.

This movement of the hot-spots is responsible for the confusing values seen in the *Iteration* and *Process* graphs, so understanding their behavior in more detail is useful. Taking a closer look at the light yellow area of the *Linear-scale map* reveals that the values in the background are not exactly the same. There are also some patterns there, but they are not very easy to see as their differences are small compared to the range of the graph. On the *Histogram-equalized map*, light yellow and dark red still mean the same lowest and highest values as on the *Linear-scale map*, but here the histogram of the map values is equalized so that every color level is used for approximately the same number of data points. This produces the maximum contrast on the map and reveals previously invisible details: there are many more systematic details down to the finest granularity than there are visible on the *Linear-scale map*.

Figure 3.6 shows more metric graphs of the test case execution, including the time taken by different activities and the number of bytes transferred in each timestep. Where no explicit legend is given, the different colors have the same meaning as before.

The *Execution Time Breakdown* graph (upper left) uses a different coloring where turquoise is the average time each process spent in pure computation (i.e. non-MPI functions), the small magenta part is the useful time spent in MPI communication, orange is the blocking time in situations such as late sender, and black is synchronization time at barriers. Together, these values make up the total execution time of each timestep, averaged over all processes. *PEPC* iteration execution time does not show any differences between processes as collective synchronizations and communications synchronize the processes in every timestep.

The high peak values every 100th iteration are due to checkpointing. Also notable on this graph is the evolution of the total execution time per iteration along the 1,300 timesteps, gradually increasing more than twofold from around 5.5 seconds to more than 12 seconds. From the breakdown this is seen to be due to the computational workload itself growing over time (as the pure computation time is growing in the same way), however, MPI blocking and synchronization times are growing as well.

On the *Execution Time Proportions* graph (upper right), the same data is normalized for each iteration to show the fraction of execution time spent in each activity. This graph shows that the proportion of pure computation time is shrinking from around 78% down to 73%, while the proportion of MPI waiting time grows from 3% to 6% and the proportion of MPI synchronization grows from 18% to 19%. MPI blocking time is therefore the fastest growing problem, even though the time spent in synchronizations is still higher.

The *Point-to-point Communication Time* graph (lower left) shows the communication imbalance very clearly, as the median and maximum times spent in point-to-point communications are much higher than the minimum, and the median is around halfway between the minimum and maximum. This means that the imbalance in point-to-point communication time involves many or most of the processes. Comparing with *Point-to-point Late Sender Time* (lower right), most of the time is actually spent in situations where the receiver is blocked, waiting for the corresponding send to be initiated, which suggests that some point-to-point messages

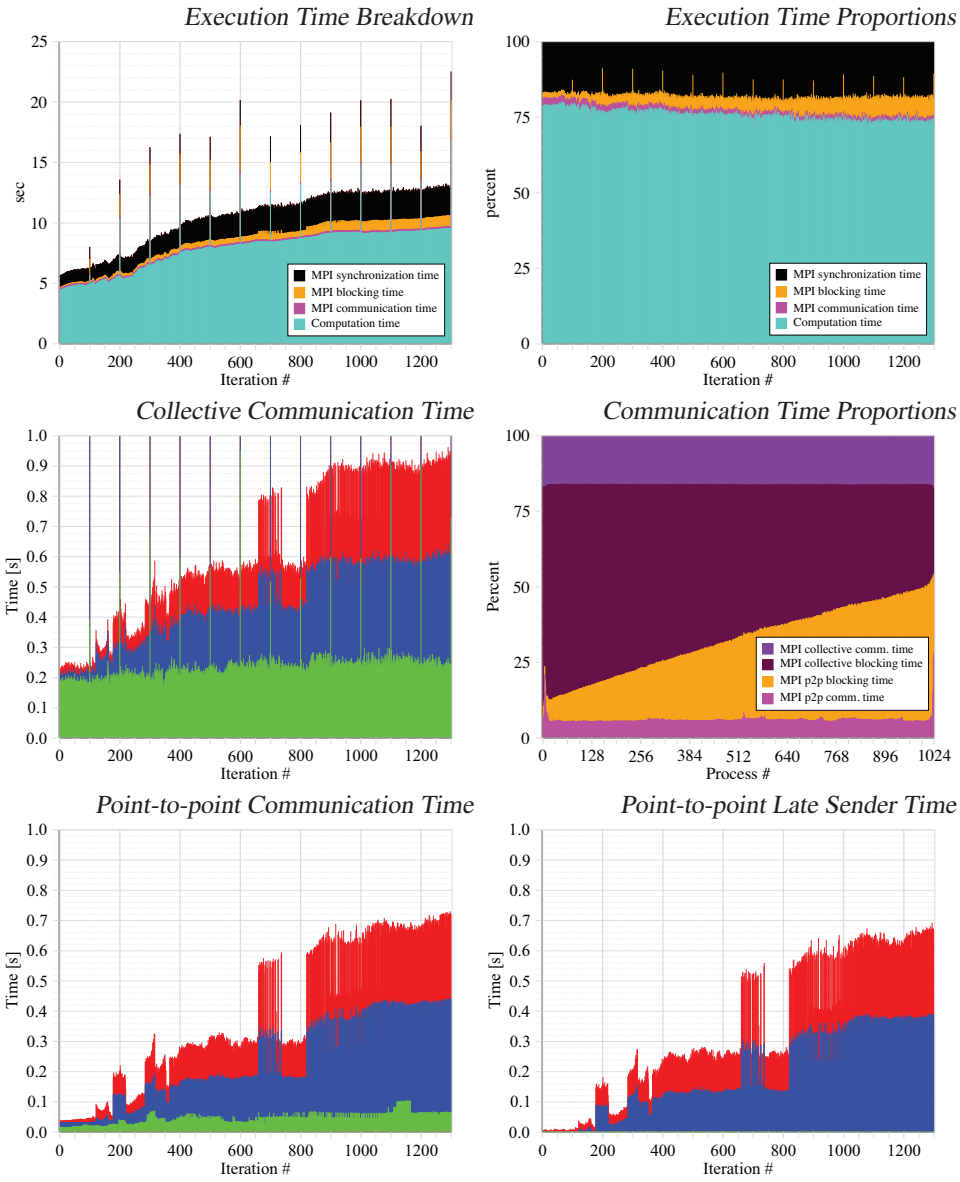


Figure 3.6: *Graphs of time and bytes transferred in different communications in PEPC.*

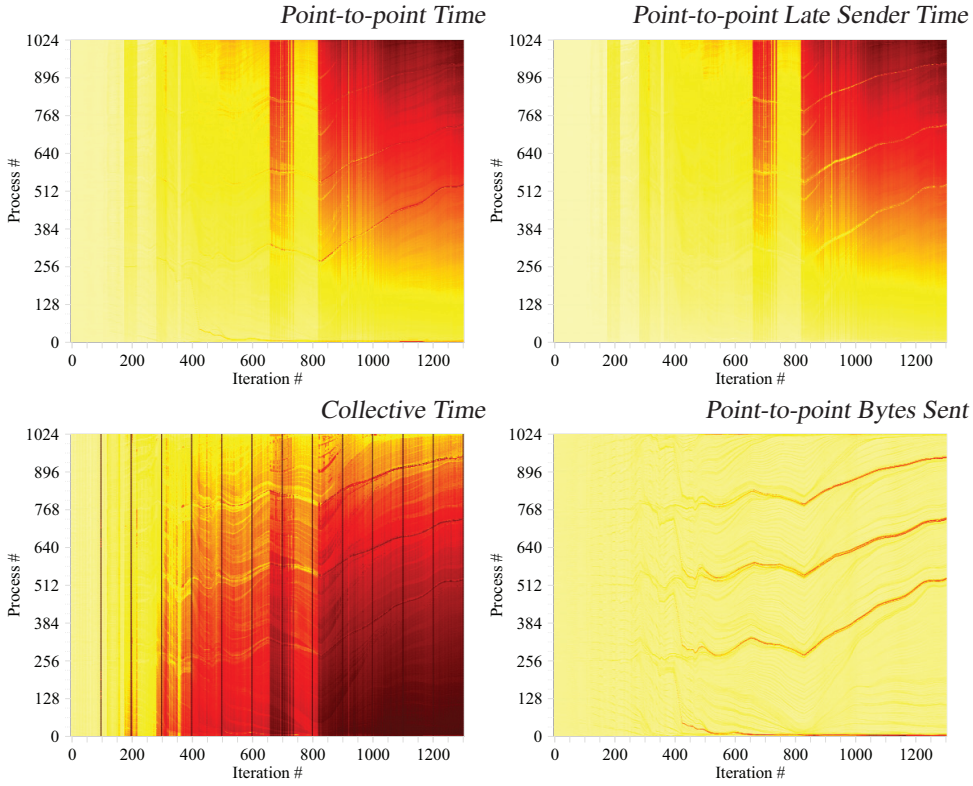


Figure 3.7: *Maps of time and bytes transferred in different communications in PEPC.*

are not sent in time. This causes a majority of processes to spend more time in point-to-point communication than is absolutely necessary (which is around the minimum value shown in green).

The *Collective Communication Time* graph (center, left) also shows an imbalance very much like that seen on its point-to-point counterpart, however, with a much higher minimum. Apparently, the minimum time for collective communication in each timestep is longer than the corresponding point-to-point time, but everything in excess of this minimum closely resembles the point-to-point late sender time. The high peaks every 100th iteration are due to checkpointing activity in those timesteps.

Figure 3.7 shows some maps that help clarify the nature of the communication time imbalance. The *Point-to-point Time* map (upper left) shows how the above mentioned communication time imbalance is distributed among the processes in a very specific and systematic way. Generally the higher the MPI process rank, the more time it spends in point-to-point communication. There are some exceptions to this rule, as the processes which send more point-to-point messages take somewhat longer than their neighbors, particularly the hot-spots which all have higher communication times.

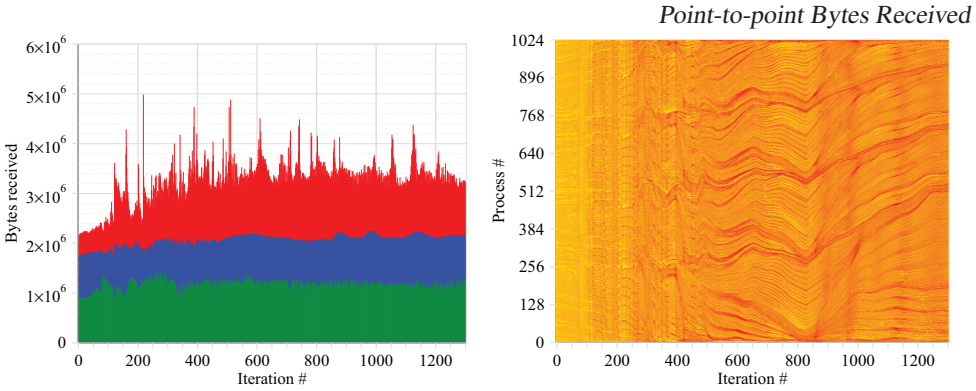


Figure 3.8: *Graph and chart of MPI Point-to-point bytes received in PEPC.*

The *Point-to-point Late Sender Time* map (upper right) shows a rather similar distribution, with one major difference being the hot-spots. These show very low waiting time in late sender situations, but they have a very high amount of time spent in point-to-point communication. This means that they really communicate a lot and then do not spend much time waiting, while all the others that must wait for them do.

The *Collective Time* map (lower left) also shows something interesting. It seems to be the inverse of *Point-to-point time*, in the sense that the order of MPI ranks is reversed here. The higher the MPI rank, the less time it spends in collective communications. This suggests that the price of the communication imbalance in point-to-point communications must be paid again in collective communications. First point-to-point communication causes the processes to get out of balance, then the next collective communication synchronizes them again, and the processes which wait less in point-to-point and are slightly ahead of the others have to wait more while the ones that are later wait less. So in the end they accumulate the same amount of blocking time, which, however, is distributed in reversed fashion across the processes.

This phenomenon is clearly shown in the *Communication Time Proportions* graph (center, right) in Figure 3.6, which shows the process ranks on the x-axis, and the proportion of MPI point-to-point communication time (magenta), point-to-point blocking time (orange), collective blocking time (maroon) and collective communication time (violet) on the y-axis. The most important aspect of this graph is the diagonal border between point-to-point blocking time and collective blocking time. The higher the MPI rank, the more point-to-point blocking time and the less collective blocking time was diagnosed, with both waiting time categories consuming around 80% of the total communication time. This highlights how seriously this problem degrades communication efficiency.

But what causes the imbalance? Examining the *Point-to-point Bytes Sent* graph (lower right) in Figure 3.7 it is clear that the hot-spot processes are sending much more data than any other process. The comparison of *Point-to-point Bytes Sent* (Fig 3.7, lower right) and *Received* (Figure 3.8) graphs reveal that while the amount of bytes received is relatively balanced through both the process and time dimension, the amount varies heavily between senders. Therefore the hot-spot processes send much more data than other processes, and they send

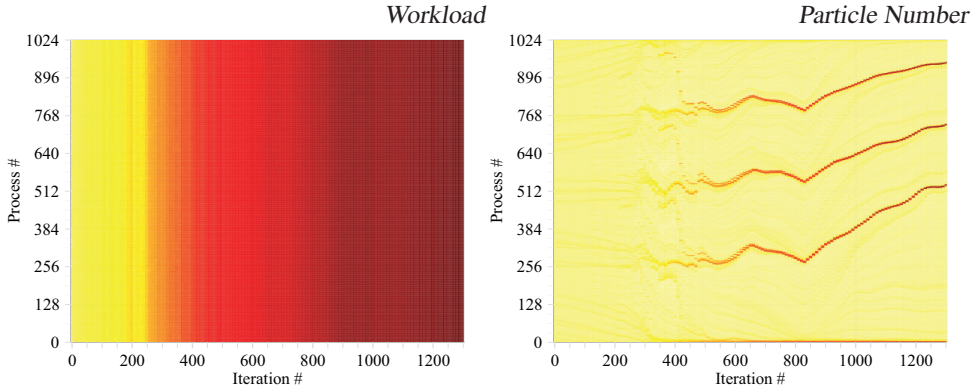


Figure 3.9: *Maps of application-specific metrics from log files written every tenth timestep.*

data to all. This suggests that the hot-spot processes are the communication bottleneck, as they send much more data than the others, and as seen in Fig 3.5, they also send many more messages than the others. Moreover, as the developers pointed out, the messages are sent in process rank order, such that higher-ranked processes have to wait for them to complete sending data to lower ranked processes before they receive any data themselves.

Figure 3.9 shows maps of two application-specific metrics extracted from *PEPC* application log files written every tenth timestep. On the *Workload* map we see the workload metric calculated by the application. According to the developers, *PEPC* runs a workload-balancing algorithm every timestep where it balances this metric, therefore it is no surprise that it really is balanced over the processes. It further shows some growth over time which correlates with the growing computational part of the execution time. On the *Particle Number* map we see the number of simulated particles assigned to each process. This means that the workload-balancing algorithm assigns a very large number of particles to the hot-spot processes which in turn causes a communication bottleneck to appear on those processes and leads to the communication imbalance.

The TAU Paraprof 3D visualizer [58] is a third-party tool able to visualize Scalasca analysis reports. As illustrated in Figure 3.10, 3D visualization is a promising complement to 2D maps, as it can be easier to compare scales of data at different (iteration, process) coordinates by comparing the height of bars as opposed to comparing the brightness of colors.

3.3.2 Conclusion

In our analysis of the *PEPC* execution on JUGENE we have identified some complex performance patterns, and have found a good potential for communication performance improvement. The *PEPC* developer team gained valuable insight from our analysis, and they have since been able to confirm our findings about the serious imbalance in particle numbers that we identified as the root cause of the point-to-point communication problem. They have conducted a significant rewrite of their algorithms since our analysis, enabling them to avoid this problem.

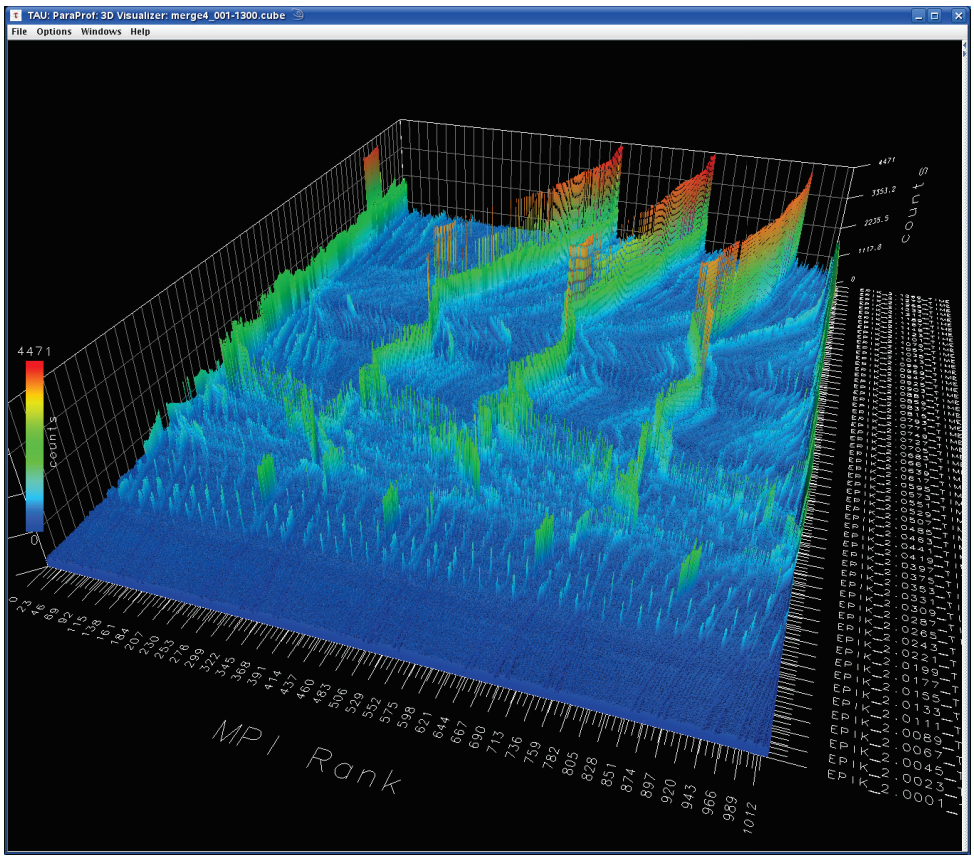


Figure 3.10: *Point-to-point sent message count in the Paraprof 3D visualizer.*

The depth of analysis we conducted in this case would not have been possible without our Scalasca extensions, namely iteration instrumentation and the different visualization techniques used to make the huge amounts of data collected accessible to the user. During the course of the analysis, we have found all the different kinds of visualization techniques (iteration and process graphs, linearly-scaled and histogram-equalized 2D maps, 3D visualization) useful in many ways, as the insights they provide often complement each other.

We have also found that there are serious limitations concerning the visualization of the huge amounts of data collected, as it can easily happen that on the monitor or printer used for displaying the data it is not possible to get sufficient resolution to have one pixel for each application process or iteration. Other groups also identified this issue [53, 11] and found different solutions, such as zooming in on the visualized data while also displaying a miniature map of the whole chart marking the zoomed region, or letting the user choose from different display options when there is more than one data point for a single pixel, such as the maximum, minimum, mean, or median of the values. These techniques can provide partial solutions to the problem, but further investigation of this topic could also prove to be valuable in the future.

Visualizing time-dependent behavior with an animation, where the user can step through iterations to track changes of a metric over time, also seems to be an interesting possibility that proved useful in understanding *PEPC* measurement results. Furthermore, 3D visualization such as the one offered by TAU was found to be extremely valuable and perhaps the most insightful of all the different visualization techniques, but also the technically most challenging due to problems such as very large polygon counts required to completely visualize large datasets. Further research and development work in the performance visualization area is expected to yield valuable results but falls outside the focus of this thesis.

The most important lesson learned from this study is that sometimes, especially in the case of applications using dynamic load balancing, looking at an overall call-path profile of the application might simply not be enough. To understand certain performance issues, we have to understand the evolution of the application's performance behavior over time. We found that time-series profiling, combined with good visualization techniques, taking the iteration as the natural building block for characterizing application performance is an invaluable tool for gaining new insights into the performance of such applications.

3.4 Summary

As we have shown in this chapter with additional details in Appendix B, temporal variations are wide-spread and may appear in highly diverse patterns ranging from gradual changes and sudden transitions of the base-line behavior to both periodically and irregularly occurring extrema. The SPEC MPI applications provide a wealth of examples showing the variety of performance dynamics patterns. To complicate matters further, the temporal behavior can be subject to significant process-dependent variations, that is, the performance behavior can be a function of both time and space. Recognizing the relationship between temporal and spatial patterns can be crucial for the understanding of performance problems, as shown by the example of the *PEPC* Coulomb solver, which was found to suffer from a gradually increasing communication imbalance caused by a small group of processes with time-dependent

constituency, as shown in Section 3.3. Sometimes, behavioral deviations may also be caused by outside influences, such as operating system jitter or application interference, which may limit their reproducibility.

Deferring further investigation/consideration of iteration analysis presentation and visualization challenges, the next chapter will address the analysis storage which grows linearly with the number of iterations and would cause problems in the case of long-running applications consisting of thousands and sometimes millions of iterations. Instead of storing every iteration separately, we introduce a novel, clustering-based lossy compression algorithm tailored to the specific problem of compressing iterative call-path profiles. This compression method offers an easily configurable tradeoff between storage requirements and compression quality, enabling the collection of high quality data at a fraction of the original memory footprint.

Chapter 4

Compression of Time-series Profiles

Writing a separate call-path profile for every iteration poses scalability problems in terms of the number of iterations that can be captured and analyzed. The most severe restriction arises from the buffer space required to store such a time-series call-path profile measurement because adding a time dimension multiplies the amount of data by the number of timesteps. If flushing of profile data buffers is to be avoided during measurement, due to its disruptive impact on the behavior to be observed, time-series call-path profiles can exceed available buffer space — especially when the call tree is large and more than one metric is collected. We expect that this problem will be aggravated by the shrinking memory/core ratio in planned future systems, such as the Blue Gene/Q. Moreover, even if the perturbation caused by flushing can be tolerated or compensated for, the aggregate size of the profile across a potentially large number of iterations and processes may hinder post-processing and interactive end-user analysis, which often occurs on a front-end node or desktop with moderate processing and memory capacity. This is another problem expected to get worse over time, as the number of cores in typical machines and consequently in typical jobs shows a steadily increasing trend, leading to growth in file sizes in an additional dimension.

In this chapter, we present a runtime approach for the (lossy) semantic compression of time-series call-path profiles. Our approach of incremental on-line clustering of single-iteration profiles enables the collection of multi-metric call-path-level data for thousands of iterations, while consuming only a few times the space of a single-iteration profile to circumvent the need of intermittently flushing the data to disk. Exploiting repetition in the time-dependent behavior of the target application, we achieve good compression rates without sacrificing important performance details or introducing major artifacts. Since calculating the similarity between two iterations based on one or more metrics and a potentially large number of call paths may be prohibitively time consuming, we keep the runtime overhead low by calculating the separation distance based on only a condensed version of the profile data. At the same time, this measure also helps in reducing the impact of the curse of dimensionality by reducing the dimensionality of the distance operator. Moreover, to account for the fact that different processes may exhibit different temporal patterns, the compression is a purely local operation, resulting in a custom-tailored process-local partitioning of the iteration space, not requiring any communication or synchronization at runtime.

The chapter is structured as follows: we explain our compression algorithm along with our design choices in Section 4.1. In Section 4.2, we offer a quantitative evaluation of our approach based on the SPEC MPI 2007 benchmark suite in terms of (i) the accuracy of the compressed

data, and (ii) the runtime overhead incurred. Section 4.3 discusses the qualitative evaluation of the compression algorithm based on visual comparison of metric graphs before and after compression, to give some first-hand impressions about the compression quality. A detailed large-scale example using the *PEPC* application presented in Section 4.4 demonstrates the value of our solution for the performance analysis of long-running applications with significant time-dependent behavior. Finally, in Section 4.5, we summarize and discuss future work.

4.1 Compression algorithm

We introduced the Scalasca measurement library in Section 1.4.3. The standard call-path profiling techniques were extended to time-series profiling in Chapter 3. On page 30 we introduced a formal definition of these time-series profiling measurements, which we will use again in this chapter to facilitate easier discussion of the compression algorithm.

Our compression algorithm is based on the idea of *incremental clustering*. Clustering is the process of grouping a set of physical or abstract objects into classes of similar objects. A cluster is a collection of data objects that are similar to others within the same cluster and dissimilar to the objects in other clusters [39]. In our case, the objects to be clustered are the metric profiles collected for individual iterations as they are generated while the target application progresses. As the goal is to achieve the best overall characterization, iterations are considered as independent entities for clustering without regard to their sequential order. To accurately cover process-dependent variations of temporal patterns, each application process makes independent clustering decisions, which has the additional advantage that neither communication nor synchronization among different processes are required at runtime. Thus, we can restrict our discussion to the actions performed by a single process. The algorithm is lossy in the sense that the compressed data no longer contains all the information necessary to restore the original data, although we will show that the result of such a reverse transformation comes very close to the original data.

4.1.1 Clustering

With every new iteration i, a new process-local *iteration profile* $t_i : c \mapsto \vec{m}$ is created, which maps a call path onto a vector of directly measured metric values and which can be stored as a matrix $T^i_{c,m}$. The clustering occurs incrementally because every new iteration initially constitutes a new cluster. Once a predefined maximum number of clusters is reached, the two clusters closest to each other are merged. For each cluster, we store only the iteration indices assigned to it and the mean profile, which is obtained by performing an element-wise arithmetic mean operation on the constituent matrices T^i, exploiting the associativity of the mean operator when merging clusters that represent more than one iteration. Using the disjoint-set forest data structure [14] makes cluster creation and merge plus retrieval of the constituent index set an asymptotically constant-time operation as opposed to a naive linear-time implementation.

4.1.2 Distance function

Which clusters are closest to each other is determined based on the distance between cluster means. However, calculating a distance function based on entire profile matrices is inefficient and may also adversely affect the clustering quality, which is sensitive to the dimensionality of the distance function ($|C| * |M|$ if the full matrix is considered). If too large, similarities between objects may be hidden. For this reason, the distance is computed based on a condensed version of the profile matrix. The condensed version is a vector

$$\vec{e} = (M_1, \ldots, M_m, D_1, \ldots, D_d)$$

composed of an extended set of metric values aggregated across the entire iteration (i.e., all call paths), such as the total time spent in that iteration. The values $M_1, \ldots, M_m$ represent the directly measured metrics, whereas $D_1, \ldots, D_d$ represent the derived metrics, together comprising the full set of metrics listed in Table 1.1. The idea behind the condensed profile is that $\vec{e}$ still carries enough information to characterize the performance behavior of an iteration accurately enough although it no longer refers to individual call paths, thus eliminating the need to compare full matrices $T^i_{c,m}$. Instead, differences with respect to individual call paths are approximated by adding more specific derived metrics. For example, the *MPI point-to-point communication time* metric is the sum of time values on MPI point-to-point communication call paths. At the same time, the condensed profile lowers the runtime overhead by making the distance calculation much more efficient and enables more conclusive clustering decisions by drastically reducing the dimensionality of the distance function ($|M|+|D|$ usually $\ll |C| * |M|$).

The distance between two vectors $\vec{e}_1$ and $\vec{e}_2$ is then calculated using the Manhattan distance [59]. Whenever a new cluster is created from a fresh iteration profile, we calculate the distance between the new cluster and those that already exist. In addition, if the desired maximum cluster count has been reached and a merge operation takes place, the distance of the resulting new cluster from all remaining clusters also has to be calculated. Overall, this still leads to linear computational complexity $O(C)$ in terms of the maximum number of clusters C for each new iteration. Note that the distance calculation count is at most twice as much as the iteration count (every time, there is one new iteration and at most one merge). In contrast, the space complexity is $O(C^2)$ because we need to store the distance between every pair of clusters.

Since metrics may have different domains and their values may cover different ranges of magnitude, we face the question of how much weight to assign to an individual metric when calculating distances. Table 1.1 shows the full set of possible Scalasca metrics produced by summary measurements. The metrics not applicable in our test cases (e.g., OpenMP metrics) or not relevant from the compression's standpoint (e.g., *Measurement overhead*), are marked with grey background. Metrics are organized in tree hierarchies defined by subset relationships, with general metrics as tree roots and more specific metrics as leaves in the tree (e.g., total time $\rightarrow$ execution time $\rightarrow$ MPI time $\rightarrow$ MPI communication time $\rightarrow$ MPI collective communication time). On the one hand, the subset relationship implies that more general metrics have higher values than more specific metrics. On the other hand, more specific metrics are typically more indicative of performance problems, such as

communication overhead. We therefore decided to assign metrics equal weights and normalize the vector elements accordingly.

As reference value for the normalization, we chose the process-local running average for all preceding iterations because it is relatively stable with respect to noise-induced outliers. Where necessary, the average over all iterations could be obtained from a prior run, and can be expected not to vary substantially in the presence of minor run-to-run variations. The first distance calculation is performed only after the desired maximum number of clusters has been reached, therefore a fair number of iterations have passed before the current average is determined for the first time. In principle, the first distance calculation could be delayed even further, depending on the availability of memory to hold iteration profiles representing unmerged clusters. Although the use of the running average may lead to premature distance calculations, which would be hard to redo without incurring substantial runtime overhead, our results in Section 4.2 indicate that the inaccuracy resulting from this limitation has overall little influence on the fidelity of our compression. We therefore believe that in most cases a single measurement will be sufficient.

4.1.3 Emphasizing the baseline

The algorithm discussed so far has a serious shortcoming. It emphasizes noise and extrema much more than it characterizes the baseline behavior. In some cases, for example in the *Point-to-point time* metric of *143.dleslie*, which is quite noisy due to the quite low magnitude of the data, the algorithm might blur periodic low-amplitude changes in the baseline by merging their constituent clusters, while preserving a number of prominent but noise-induced extrema that are distinct enough to have dedicated clusters reserved for them. The problem occurs whenever the distance between extrema is large compared to the distances within the small-scale repetitive pattern. However, baseline changes often carry valuable information on general performance trends and are therefore desirable to keep, whereas extrema often turn out to be irreproducible noise and a minor contribution to overall performance.

To better accentuate the baseline in comparison to extrema, we exploit the fact that the number of iterations that stand out is usually much smaller than the number of iterations on the baseline level. Using a heuristic, we distort the distance metric in such a way that the distance among larger clusters becomes higher, whereas the distance between smaller clusters with only a few elements becomes lower. Two small clusters are then much more likely to be merged than two larger clusters representing many iterations. As distances are calculated whenever a new cluster is created and used for later comparisons, we have to make sure that that the values used in the heuristic at the time of the calculation are still valid when the distance value is used. Indeed, the heuristic only depends on the number of iterations associated with a cluster. This can only change when two clusters are merged, at which point the distances are also re-calculated, which means that the values used in the heuristic remain correct as long as the distance value is used. As a result, distances do not have to be recomputed until one of the clusters is merged with another one, leaving the complexity linear in terms of the maximum number of clusters.

To distort the Manhattan distance, we multiply the distance by a value that is a function of the sum of the sizes of the two clusters (i.e., the number of iterations associated with

them). The exact function applied here (chosen after trying several possibilities) is defined by Equation 4.1, where n is the sum of the sizes of the clusters.

$$m(n) = \begin{cases} 0.4 + 0.05n, & \text{if } n \leq 12 \\ \sqrt{0.4 + 0.05n}, & \text{otherwise} \end{cases} \tag{4.1}$$

At low n values, the multiplier is growing fast, while at larger clusters the speed of growth decreases so that the difference between multipliers of large clusters becomes less dominant.

4.1.4 Call-tree equivalence

The algorithm explained so far is based on an elastic distance criterion, allowing iterations with very different call trees to be merged. This property may allow the occurrence of *phantom call paths* in the compressed profile, which are call paths associated with an iteration although they have never been visited during this iteration. Since phantom call paths may lead to an invalid performance assessment, they are a serious problem and should be avoided whenever possible. Furthermore, call-tree equivalence between two iterations is often a good indicator of similar performance characteristics. For these two reasons, we only merge iterations with equivalent call trees. Call-tree equivalence between two iterations i_1 and i_2 can be defined in two ways:

- *Weak equivalence*: every call path visited in i_1 has also been visited in i_2 and vice versa.
- *Strong equivalence*: every call path has been visited as many times in i_1 as it has been visited in i_2.

Weak equivalence excludes phantom call paths, while strong equivalence also considers quantitative similarity. To enforce call-tree equivalence between clusters before they are merged, we partition the clusters into call-tree equivalence classes. Then every new iteration profile either forms a new class or is added to an existing one, depending on whether there is already a cluster whose call tree is equivalent to the new one. If the maximum number of clusters would be exceeded, the two clusters closest to each other that are located within the same call-tree equivalence class are merged.

Since call-tree equivalence is no longer an elastic criterion, the number of equivalence classes is not configurable. Especially when strong call-tree equivalence is used, the total number of clusters may exceed the predefined threshold, resulting in a large number of equivalence classes each with only a single cluster, increasing storage requirements and also potentially degrading the compression fidelity as distance-based clustering is then suppressed (every iteration in the same call-tree equivalence class is merged immediately, no matter what the distances are). On the other hand, if their number is small enough, enforcing call-tree equivalence can improve the compression fidelity with respect to call-tree structure and count-based metrics, as we will see in Section 4.2. Whether and to which degree call-tree equivalence should be enforced is therefore highly application-dependent and must be decided based on the total number of call-tree equivalence classes, which would have to be determined from a prior measurement. To avoid this extra measurement, a dynamic scheme can be employed that starts with strong equivalence and switches to weak equivalence after re-grouping the existent

clusters if the number of call-tree equivalence classes grows too large in comparison to the desired number of clusters, thus keeping the number of clusters in check.

Since in our test cases the number of weak equivalence classes exceeded 8 only in two cases, as can be seen in Table 4.1, we expect at least weak equivalence to be a viable option for most applications. Note that the data about *DROPS* is based on measurements taken using a special measurement technique, a hybrid sampling-based approach which we will introduce in Chapter 5, as collecting meaningful measurements using our normal approach is not possible in the case of *DROPS*.

To give a high-level overview of the algorithm, Figure 4.1 illustrates the basic steps of our algorithm for one process using a maximum number of four clusters. Understanding this example facilitates the discussion of the algorithmic complexity in the rest of this section.

While enforcing call-tree equivalence adds the cost of comparing potentially large call trees to determine the equivalence class of an iteration, it also reduces the number of distance calculations because distances are calculated only within each class. The performance disadvantage therefore diminishes as the number of clusters is increased. Table 4.2 presents an overview of the time and space complexity of the different steps of the algorithm. Locating a single entry in the matrix that stores the pre-calculated distances takes $O(C)$ time as the distance matrix is implemented as a hierarchy of linked lists that facilitates frequent addition and removal of rows and columns and keeps track of the lowest distance in the matrix as a side effect of stepping through the lists. A naive solution to updating class assignments after a single merge could easily become $O(I)$, where I is the iteration count, but using the disjoint-set forest data structure mentioned before brings this down to $O(1)$. The hash table used for differentiating equivalence classes has $O(P)$ comparisons for inserting a single iteration in the worst case, but it is asymptotically $O(1)$, meaning one tree traversal to generate the hash and exactly one tree comparison afterwards in most cases.

4.1.5 Reconstructing aggregate profiles

Although our algorithm performs a lossy compression, we can accurately reconstruct an aggregate profile that summarizes across all iterations. Adding up all clusters weighted by the number of iterations they represent will yield an exact profile without any errors introduced by the lossy compression, as if obtained without iteration instrumentation. While small differences between an original and a reconstructed summary profile may be caused by compression overhead, compression errors manifest only when considering individual iterations.

Table 4.1: *Application characteristics: execution times and iteration, call-path and call-path equivalence class counts.*

Application	Execution time (s)	Iterations	Call paths total	Call paths per iteration min.	Call paths per iteration avg.	Call paths per iteration max.	MPI per iteration min.	MPI per iteration avg.	MPI per iteration max.	Call-tree equiv. classes weak	Call-tree equiv. classes strong
121.pop2	461	440	49,267	105	111.0	124	17	20.0	24	3	**10**
125.RAxML	630	21268	191,237	3	9.0	15	1	2.0	4	**18**	10,641
126.lammps	574	300	11,442	33	36.8	62	12	13.3	22	**8**	59
128.GAPgeofem	575	2501	57,557	23	23.0	24	8	8.0	8	2	**11**
129.tera_tf	281	190	2,170	11	11.1	22	4	4.0	8	2	**3**
132.zeusmp2	247	200	25,657	127	128.0	128	62	62.0	62	2	**2**
137.lu	269	180	2,940	16	16.0	18	7	7.0	8	2	**3**
143.dleslie	251	15054	271,776	18	18.1	22	3	3.0	5	2	**4**
145.lGemsFDTD	326	1500	84,274	56	56.0	58	3	3.0	4	2	**2**
147.l2wrf2	867	720	192,075	263	265.2	618	11	11.0	34	**6**	495
DROPS	5,841	250	178,736	685	714.9	850	79	79.9	97	**3**	196
PEPC	497	1300	66,253	50	50.9	65	28	28.5	35	**4**	1298

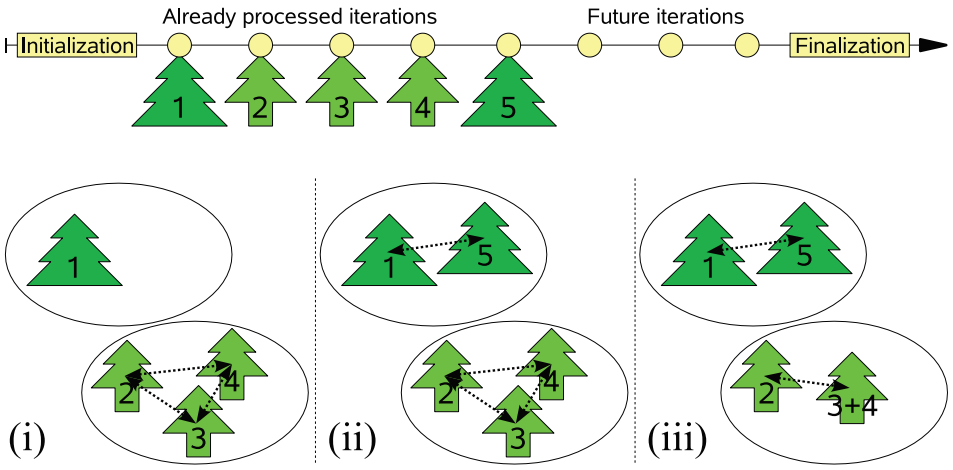


Figure 4.1: *Incremental on-line clustering of iteration call-tree profiles into a maximum of four clusters. (i) The call-tree profiles for the first four iterations are stored directly, yet divided into two equivalence classes. (ii) The call-tree profile for iteration 5 is matched to the equivalence class of iteration 1. (iii) The pair of clusters with the shortest separation distance (here 3 and 4) are merged to retain only the desired number of clusters.*

4.2 Evaluation

For evaluation purposes, we use the same applications as in Chapter 3, the SPEC MPI 2007 applications with 'lref'-sized datasets on JUROPA and the *PEPC* application on JUGENE. The application *142.dmilc* is not used due to its complicated nested loop structure involving `goto` statements, which does not fit our model completely. Also, *142.dmilc* does not have a main loop, but five different loops at the highest level taking about the same amount of time. While handling this application would not be impossible, it would require much more elaborate handling, providing little benefit at the expense of complicating the chapter. Execution characteristics of the applications are summarized in Table 4.1. The *PEPC* application will be considered separately in Section 4.4.

In addition to the wall-clock execution time and number of iterations (or timesteps) in the main computational loop of each application, full summary measurements with instrumentation distinguishing each iteration allow analysis of the call-tree sizes, which is included in Table 4.1. Call paths for each iteration are further distinguished to consider only those ending with MPI communication and synchronization functions. Minimum, average and maximum call-path count statistics are also calculated. (The number of call paths included in a summary profile depends on function in-lining by the compiler and filtering applied during measurement [86].) Furthermore, by enforcing call-tree equivalence among the sets of call paths, the number of classes resulting from weak and strong equivalence were determined. Where strong equivalence resulted in an excessive number of equivalence classes, weak equivalence was used instead (i.e., for *125.RAxML*, *147.l2wrf2* and *PEPC*). The SPEC MPI2007 benchmark suite and the *PEPC* application have a rich variety of execution and call-tree characteristics, as detailed in Chapter 3 and Appendix B.

Table 4.2: *Overview of the compression algorithm's complexity.*

# of clusters in equiv. class	C
# of metrics	M
# of equivalence classes	E
# of call paths per iteration	P
distance matrix size	$O(C^2)$
time to compute a distance	$O(M)$
time to locate an entry in matrix	$O(C)$
time to update distance matrix	$O(MC + C^2)$
cost of a tree comparison	$O(P)$
# of tree comparisons	$O(1)$
time to update class assignments	$O(1)$
time to update equivalence classes	$O(P)$

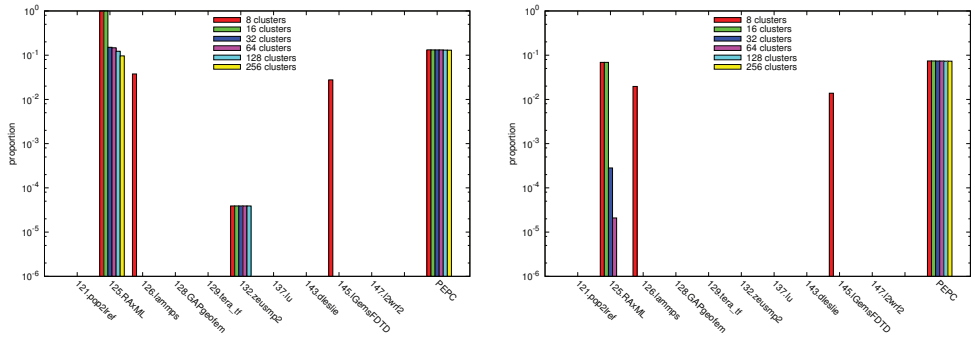
Based on the numbers of iterations and call-tree equivalence classes found in these application executions, we chose to compare results using cluster counts that are powers of two from 8 to 256. Eight clusters offer a relatively small storage overhead but require aggressive compression, and can be expected to only coarsely represent complex execution characteristics. On the other hand, 256 clusters should provide improved fidelity, but at a corresponding storage cost factor for the results.

4.2.1 Off-line version

Exact evaluation of the quality of the compressed data based on the on-line approach is not possible by simply taking measurements with and without compression and comparing the results. In that case, differences caused by run-to-run variation cannot be distinguished from those caused by the lossiness of the compression. Additionally, accurate time and memory usage measurements of the algorithm can not be taken while it runs together with the actual application measurement. Therefore, our first approach for the evaluation, as a proof of concept for the overall technique is based on an off-line version of the algorithm which works on previously collected, non-compressed measurement results. The compression algorithm is applied to this input data to create virtual "measurement results" which are equivalent to those that would be collected by the on-line version of the algorithm. These off-line compression runs were performed on the same system where the original measurements were collected to make timings comparable. The effectiveness of different configurations of the compression algorithm can also be readily compared, as they are all based on the same real measurement data. After providing proof of concept using the off-line evaluation, we present an evaluation of measurements performing compression on-line at run time in Chapter 6.

4.2.2 Quality assessment

A variety of quality characteristics were investigated for *PEPC* and the SPEC MPI2007 applications with different numbers of clusters.



(a) Call paths added. (b) MPI call paths added.

Figure 4.2: *Proportion of erroneously added 'phantom' call paths in reconstructed profiles when call-tree equivalence is not enforced during compression.*

Erroneous call paths

All execution call paths are accurately captured when call-tree equivalence is enforced, however, without it call paths can be erroneously associated with iterations where they do not actually occur by merging two iteration clusters with different call trees. Figure 4.2 shows that there are indeed significant numbers of 'phantom' call paths added for *125.RAxML*, *132.zeusmp2*, and *PEPC*. For *126.lammps* and *145.lGemsFDTD* this only occurs when using 8 clusters, and 16 or more clusters are enough to avoid this problem in those cases. The remaining six applications never had 'phantom' call paths introduced. Although many of the 'phantom' call paths are not MPI call paths, and therefore less of a concern, 'phantom' MPI call paths are found for *125.RAxML*, *126.lammps* and *145.lGemsFDTD* with smaller numbers of clusters, and predominate for *PEPC* at all cluster counts. It is important to note that these graphs are on a logarithmic scale. Since the call trees often differ, and can differ significantly especially at low cluster counts, enforcing call-tree equivalence is therefore essential for accurate time-series call-path profiling. For this reason, call-tree equivalence is always enforced in the remainder of our evaluation with the specific equivalence relation (i.e. weak or strong) determined based on the number of resulting classes, as indicated by the bold numbers in the rightmost two columns of Table 4.1. In most cases, we were able to apply strong equivalence, but for *125.RAxML*, *126.lammps*, *147.l2wrf2* and *PEPC* we had to resort to weak equivalence.

Error rates for entire iterations

Figure 4.3 shows the comparison of the average error rates with and without call-tree partitioning, for both the mean and the maximum metric values for entire iterations. These mean and maximum values are the blue and red columns respectively in the iteration graphs such as Figure 3.1(b). First of all, for every application we get better or equal results when using call tree equivalence classes showing that enforcing call-tree equivalence does not significantly reduce compression quality, and generally helps to achieve a better representation of the data.

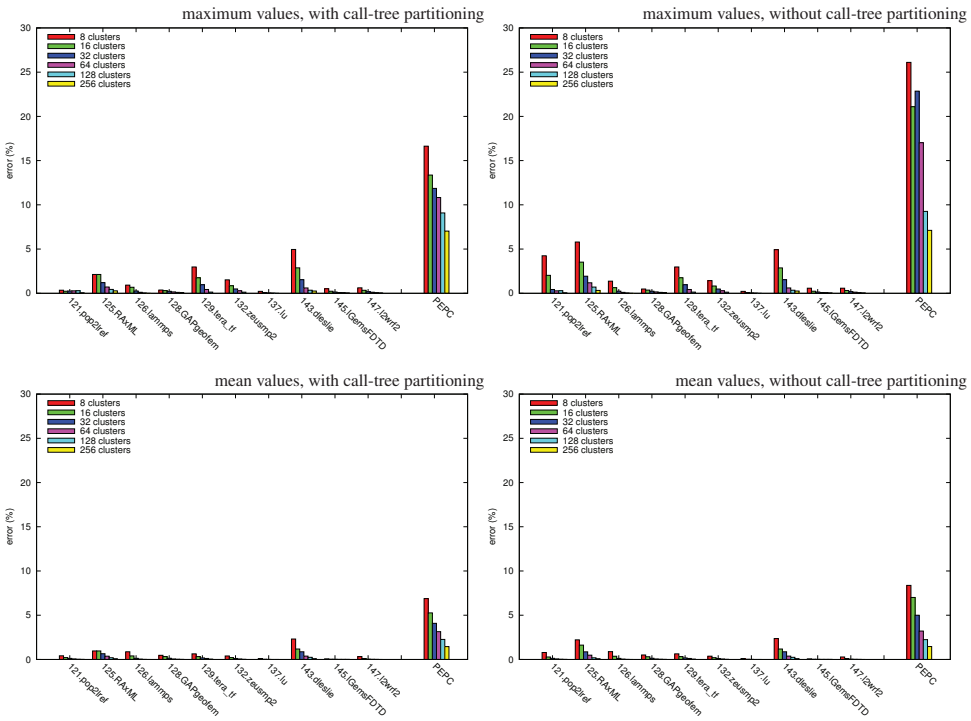


Figure 4.3: *Average error rates of the maximum and mean metric values for entire iterations, of all metrics, with and without call-tree partitioning.*

This is because in many cases call-tree partitioning has a high correlation with the changes appearing in many of the metrics, and applying call-tree partitioning gives a boost to the quality of the resulting graphs. This only makes a real impact at low cluster counts, as at higher counts the basic metric-based clustering algorithm also identifies these changes correctly, and the difference between the quality of the two methods diminishes.

We see exactly the same error rates with call-tree partitioning for 8 and 16 clusters in the *125.RAxML* case, as it has 18 call-tree equivalence classes even in the weak equivalence case, and call-tree partitioning takes precedence over metric-based clustering. When the number of allowed clusters is set to a lower value than the call-tree equivalence class count, every equivalence class is created (even if there are more of them than the allowed maximum), and no metric-based clustering takes place. As soon as we reach 32 clusters, the allowed cluster count is larger than the number of call-tree equivalence classes, and metric-based clustering takes place again, improving the quality of the compressed data.

Considering the error rates at 64 clusters, the highest value of the average error of the mean graphs without call-tree partitioning is 0.38% (*143.dleslie*) and under 0.07% for 8 out of the 10 SPEC MPI2007 applications. With call-tree partitioning the highest value is 0.48% and under 0.12% for 8 of the 10 applications. For the maximum graphs the highest values are more than doubled, but 7 of the 10 applications are still under 0.29% in both cases. Error rates

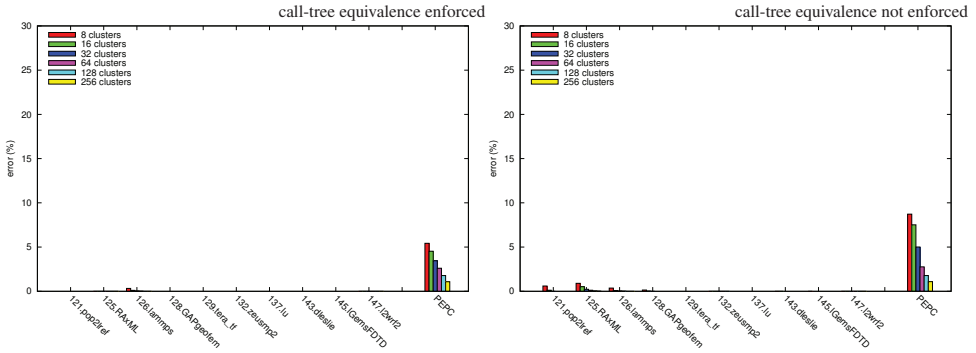


Figure 4.4: *Average error rates of the mean values of the count-based metrics for entire iterations, with and without call-tree partitioning.*

are somewhat higher when applying call-tree partitioning, as it leaves a slightly lower number of cluster merging decisions to be made based on metric-based distance computations. Still, the additional guarantees provided by call-tree partitioning easily make up for these small differences.

It is much easier to get good results for some metrics than for others. In particular, the count-based metrics, such as *Visit count* and *Bytes transferred*, are usually much more reproducible than time-based metrics, as they are not subject to noise and small measurement variations, and their changes often coincide with changes in iteration call-tree structure. In fact, for 7 out of 10 applications we get perfect results with call-tree partitioning (Figure 4.4) even at the lowest cluster counts, and the error rates of the remaining three are also very low (under 0.04% at 64 clusters). Without call-tree partitioning we only get perfect results for 2 of the 10 applications at the lowest cluster counts, which grows to 6 at 64 clusters. This means that while there is no guarantee of perfect results, for most of the applications the graphs based on the compressed data for the count-based metrics are extremely reliable, especially when using call-tree partitioning.

Error rates for individual call paths

To evaluate the quality of the metric data for each call path within each iteration, a different method is employed. Figure 4.5 shows the average error rates in the value of time-based metrics for each non-zero call-path excluding its children (i.e., the exclusive value), with and without call-tree partitioning. To emphasize the most significant call paths, error rates for each metric are normalized using the maximum value of this metric across all iterations. There are only slight differences between the two modes here, mainly at low cluster counts, where using call-tree partitioning sacrifices clusters for correct call-tree partitioning and has less clusters to use for metric-based clustering (the more equivalence classes prescribed by the call-tree partitioning, the less freedom for the clustering algorithm in its merging decisions). In both cases every application has an error ratio below 0.7% even at the lowest cluster counts, and 5 out of 10 have an error ratio lower than 0.11%. The worst case is *129.tera_tf*, where the quality is relatively low at low cluster counts, and grows only when increasing the cluster count. This

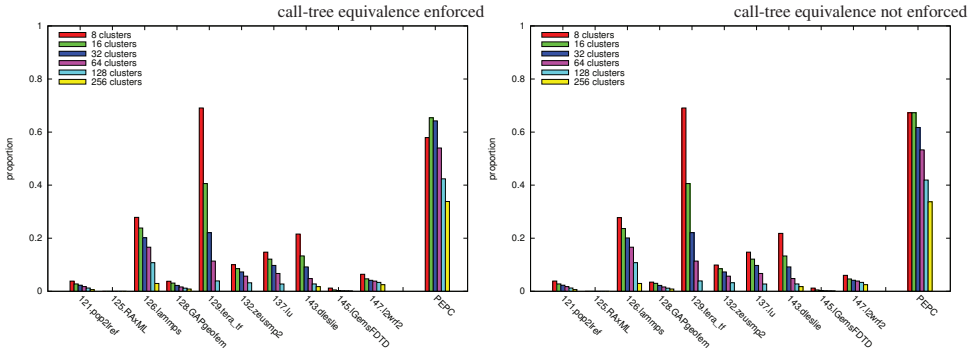


Figure 4.5: *Average error rate of time-based metrics for individual call paths, with and without call-tree partitioning.*

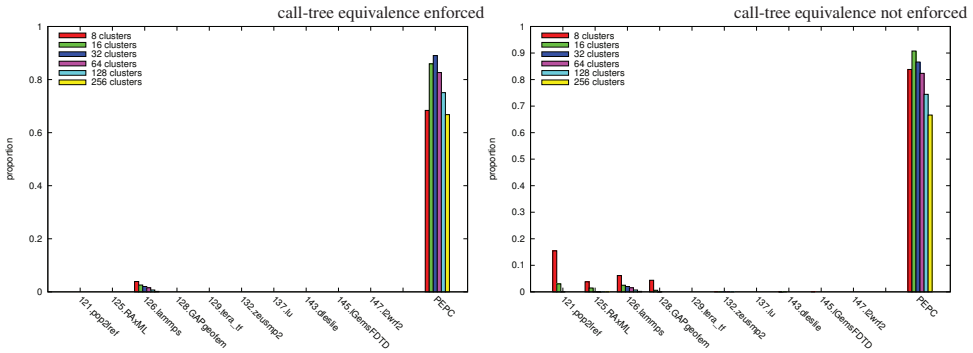


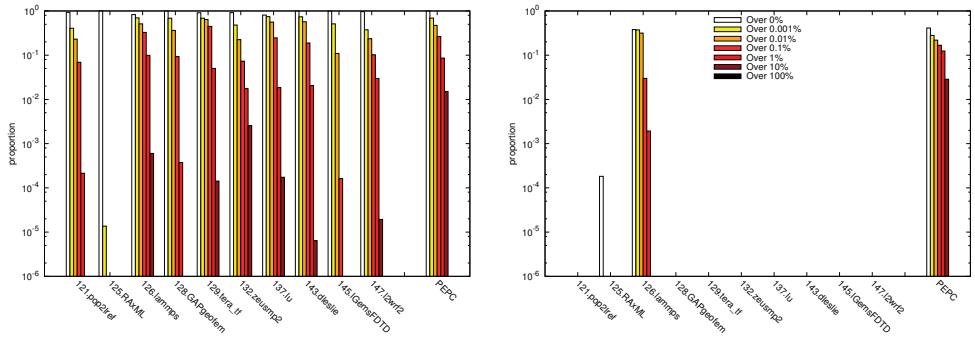
Figure 4.6: *Average error rate of count-based metrics for individual call paths, with and without call-tree partitioning.*

is because this application has a quite complex continually changing behavior, as evident in Figure B.7 on page 163, and so it is a much more challenging case for our approach than most of the other applications.

Looking at the error rates of the count-based metrics in Figure 4.6, we see that all applications that were perfectly represented at the iteration level are also perfectly represented at the deeper, call-tree level. Even without call-tree partitioning, the error rates are below 0.16% even at the lowest cluster counts. It is also notable that the count-based metrics are perfectly represented for the challenging *129.tera_tf* application.

Quantized call-path error rates

Where the very existence of call paths is false, their values are considered as error in all metrics at the call-path level, however, some amount of error is also inherent in the metric values reconstructed for every true call path from the compressed representations. For each call path, the magnitude of the error in the metrics can be determined from the difference of metric



(a) MPI time-based metrics. (b) MPI count-based metrics.

Figure 4.7: *Proportion of call paths in profiles reconstructed from 64 clusters having quantized error rates for MPI time-based and count-based metrics.*

values between the full-fidelity and reconstructed call-path profiles. The absolute differences for each metric have been normalized using the corresponding maximum exclusive call-path value, before they were averaged and histogrammed in buckets for orders of magnitude: Figure 4.7 summarizes the results for the case using 64 clusters. (Call paths which actually have zero-valued metrics are not included in this comparison to avoid dilution.)

As an example of interpreting the graph, in the case of *121.pop2lref*, the first white bar in the time-based metrics shows that nearly all call paths have more than zero error, the yellow bar indicates that around half of the call paths have more than 0.001% error, the orange bar shows that around a quarter of the call paths has more than 0.01% error, the first red bar shows that a bit more than 5% of the call paths have more than 0.1% error, the second red bar tells us that around 0.025% of the call paths have more than 1% error. The absence of the darker bars in this case indicates that no call path had more than 10% error. The absence of any bars for this application in the MPI count-based metrics graph indicates that no call path had any error in those metrics.

With 64 clusters, only two of the SPEC MPI applications have any error in their count-based metrics, one of which is *125.RAxML*, where the errors are all less than 0.001%. For *126.lammps* around one in every five hundred call paths have errors exceeding 1%, and with *PEPC* around one in every forty call paths have more than 10% error. For time-based metrics, entirely error-free reconstructions are rare, and call-path error rates over 10% for the time-based metrics are found for *PEPC* and 6 out of 10 SPEC MPI2007 applications. Such large errors typically affect less than one in every thousand call paths in these applications, with MPI call paths somewhat more frequently concerned. The largest call-path errors are found to have a magnitude of 48% for *128.GAPgeofem* and 100% for *PEPC*. Except for *125.RAxML* and *145.lGemsFDTD*, all applications have call paths with more than 1% error in the time-based metrics, which affects one in ten of *126.lammps* MPI call paths. Larger numbers of clusters can be employed to reduce the magnitude and frequency of these errors.

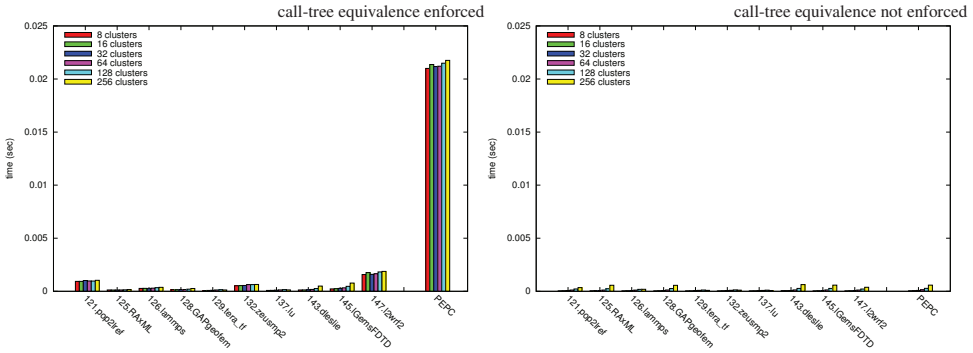


Figure 4.8: *Average time to perform the compression of a single iteration.*

4.2.3 Processing time

Comparing the average time required to perform iteration clustering with and without call-tree partitioning (Figure 4.8) shows that including it takes much longer. For the SPEC MPI 2007 applications on JUROPA, with call-tree partitioning it reaches an average value of 0.0019s per iteration in the slowest case (*147.l2wrf2*), while without call-tree partitioning they are all under 0.00064s even at the higher cluster counts, and under 0.00015s for 64 clusters. This also explains why there are not very large differences between the processing times of different cluster counts for any given application, as the call-tree partitioning part always takes around the same amount of time, and the differences that come from the other parts of the algorithm are nearly negligible compared to this. Without call-tree partitioning, the much lower clustering times show an $O(C^2)$ increase rate. While this could pose a scalability issue at higher cluster counts, the intent of the method is to prefer low cluster counts, and even at the relatively high value of 256 clusters, the time spent in distance computations is acceptably low. In some cases, however (*125.RAxML*, *128.GAPgeofem* and *143.dleslie*), it surpasses the amount of time used with call-tree partitioning included. This is because with call-tree partitioning, the algorithm does not have to calculate distances between clusters in separate call-tree equivalence classes. This optimization makes the compression time with call-tree partitioning grow much more slowly with the cluster count than without the partitioning.

In the cases of *129.tera_tf*, *132.zeusmp2* and *137.lu*, the 256 cluster case requires less processing time than the 128 cluster case, since those applications have less than 256 iterations (*129.tera_tf*: 190, *132.zeusmp2*: 200, *137.lu*: 180). When every iteration is preserved, no merging operation takes place and calculated distances never have to be updated.

More important than the absolute time of the compression operation is its duration relative to that of a single iteration. Figure 4.9 shows how much overhead the algorithm introduces into the run time of the application. When not using call-tree partitioning, 8 of the 10 applications are under 0.29% even at the highest cluster counts, with *125.RAxML* and *143.dleslie* being the exceptions. As Table 4.1 shows, these applications combine a high iteration count (and correspondingly fast iterations) with a large number of call-paths (which means more data to deal with for the algorithm), however, even these applications are under 0.25% for 64 clusters. By far the worst case is *125.RAxML* with nearly 17% clustering overhead.

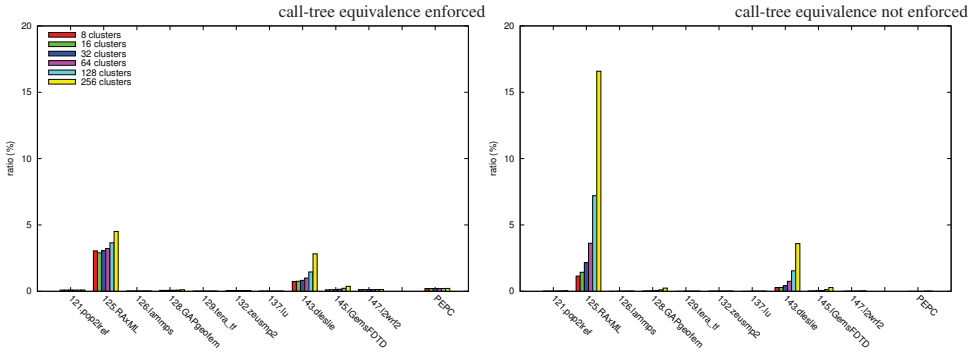


Figure 4.9: *Average compression time as a proportion of iteration execution time.*

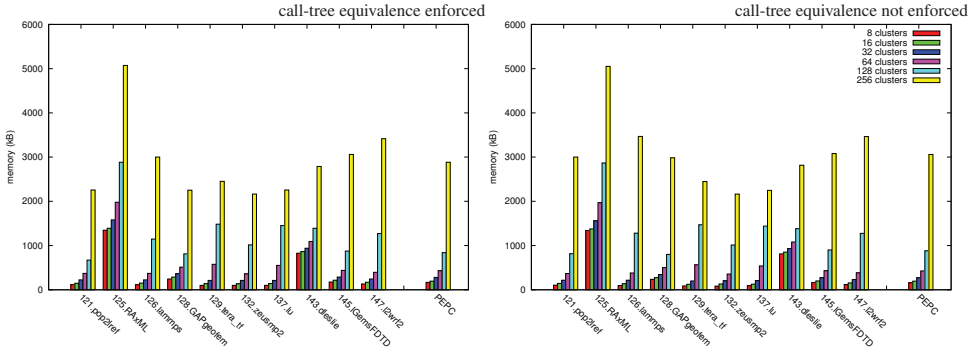


Figure 4.10: *Memory usage of the compression algorithm for different numbers of clusters.*

With call-tree partitioning included, the processing overhead for 8 of the 10 applications is under 0.38%. The worst cases are *125.RAxML* and *143.dleslie*, the applications with the very short iterations. Notably in both cases, but especially in the *125.RAxML* case, using call-tree partitioning actually lowers the overhead significantly at higher cluster counts, as only the distance calculations between clusters in the same call-tree equivalence class have to be calculated. Overall, this means that we can safely use call-tree partitioning in every case without having to fear extreme compression overhead (generally much less than 1%, in case of very short iterations somewhat more). For *PEPC*, dilation due to compression when using call-tree partitioning is much more, but still negligible compared to the average iteration time.

4.2.4 Memory requirements

Figure 4.10 shows the memory consumption of the algorithm. Most of the memory is used for storing precomputed cluster distances, so there is an $O(C^2)$ growth with the cluster count. With call-tree partitioning, only the distances between clusters of the same call-tree equivalence class have to be computed (and stored), which makes this growth somewhat slower for applications with many call-tree equivalence classes. Even with cluster sizes as

large as 256, the memory requirements are less than 6MB and therefore negligible where memory available per process is typically 512MB or more.

4.2.5 Comparison with PEPC

Comparing the error rates of the *PEPC* graphs to those of the SPEC MPI2007 applications on Figure 4.3, it is clear that *PEPC* has a higher error rate at the lowest cluster counts than most of the applications, but more importantly, its error rate does not shrink as fast as for the other applications, having notably high values even at 256 clusters, especially in the graphs of the maximum values. This is because *PEPC* has a constantly changing behavior in all its metrics, which is much harder to characterize. Also, representing the maximum value (the red part of the graphs like Figure 3.1(b) on page 32) exactly is difficult, as a single outlier on any of the 1024 processes ruins the maximum for any given iteration. As *PEPC* is an adaptive code, its count-based metrics also show constantly changing behavior, and are not much easier to characterize than the time-based metrics. This leads to the situation that while all the SPEC MPI2007 applications have a negligible error rate on their count-based metrics, in the *PEPC* case the count-based metrics have an only slightly better error rate than the time based metrics. Enforcing call-tree equivalence clearly gives an advantage when characterizing the count-based metrics, though — even in such a difficult case.

The average error rates of the time-based metrics for individual call paths (Figure 4.5 on page 59) are among the worse examples. With respect to count-based metrics, however, it is the worst example and not substantially better than with respect to time-based metrics, but still clearly acceptable, as we will see in the visual evaluation section.

4.3 Visual evaluation

Although quantitative evaluation of the compression quality is very important, in the end it largely comes down to what the user will actually infer from the difference. As in the case of MP3 music files, it is also true here that if from a practical point of view the original and the compressed representations appear identical, the applied lossy compression can be considered good enough. This is why it is important to also evaluate the quality visually, to get first hand impression of the compression quality. Figure 4.11 and Figure 4.12 give such an overview, using the *MPI Point-to-point communication time* metric as a non-trivial example. Note that the compression algorithm is trying to achieve low error rates for all metrics at the same time, not just this one, but it would not be practical to show all graphs and charts from any given measurement. Also, since *125.RAxML* does not have any point-to-point communication, we use the *MPI Collective communication time* metric there instead.

In every case, we show a comparison of a graph based on the original measurement and a reconstructed graph after the lossy compression. The two graphs are based on the same measurement, so ideally they should match completely. For every application, we chose a cluster count that is a borderline case between acceptable and good. Choosing any higher cluster counts would just make it even harder to see any difference between the original and

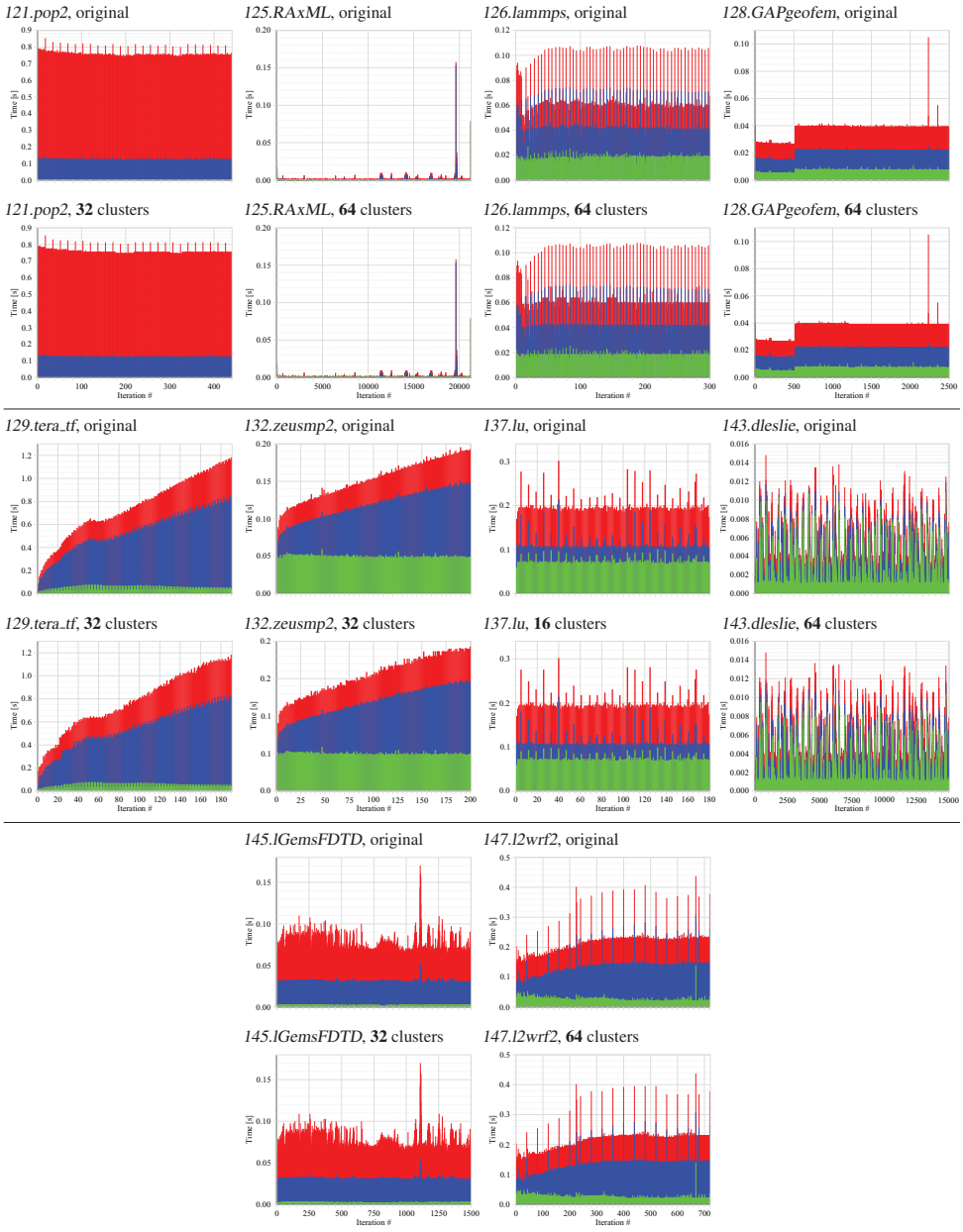


Figure 4.11: *Comparison of original and reconstructed clustered iteration graphs of communication time in the SPEC MPI applications.*

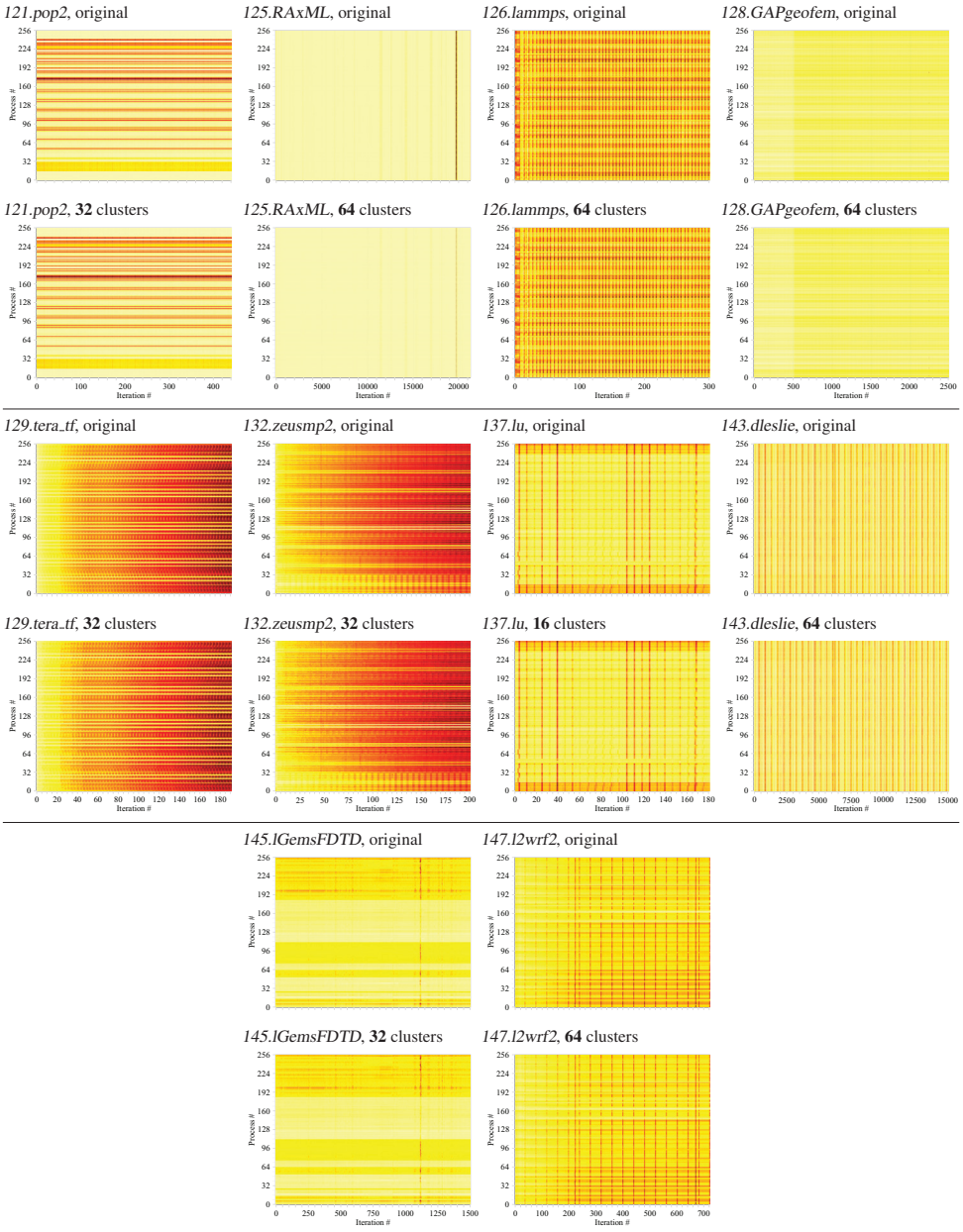


Figure 4.12: *Comparison of original and reconstructed clustered iteration charts of communication time in the SPEC MPI applications.*

reconstructed graphs. Overall, we get quite good similarity with a maximum of 64 clusters in every case. It is also interesting to note that getting a visually good match on value maps is generally easier than on the graphs. Although the color maps show more data, the graphs are the harder cases, as they exaggerate one-off outliers, especially in the maximum.

4.4 Detailed Example: PEPC

As a large-scale example, we return to the *PEPC* Coulomb-solver run with 1024 processes on JUGENE, in which we were able to find the root causes of a serious performance problem using our iteration-instrumentation approach, as discussed in section 3.3. *PEPC* is much harder to characterize than the previously considered SPEC MPI2007 applications, in that its performance is completely different on each process, its baseline is continually growing for many metrics, and the few bottleneck processes whose communication differs significantly from all others are constantly relocated by the computational load-balancing algorithm.

It is therefore no surprise that the quality analysis and comparison in in the previous section show *PEPC* to be much more susceptible to 'phantom' call paths (Figure 4.2) when not enforcing call-tree equivalence, especially 'phantom' MPI call paths, and substantial error rates requiring 256 or more clusters for reasonable accuracy of both entire-iteration aggregate metric values (Figure 4.3) and individual call-path exclusive metric values (Figs. 4.5 and 4.7). On the other hand, even with 256 clusters, processing overheads are reasonable (Figure 4.9).

Figure 4.13 gives an impression of the quality of the data produced by different maximum cluster counts. It shows the iteration graphs and value maps of two of the key metrics, *MPI point-to-point communication count* and *time*, reconstructed from compressed data with different cluster counts, and compared to the full-fidelity originals. This shows that *PEPC* is a good example of an application where the count-based metric graphs and value maps (in the leftmost two columns) are complex, yet the reconstructions are surprisingly good and with 32 clusters all the main features of the originals are already present.

MPI point-to-point communication time metric graphs and value maps (in the rightmost two columns), however, show that even 64 clusters is insufficient to reconstruct this application execution behavior. With 128 clusters it is relatively close, especially the average values, but clearly 256 clusters are needed for a really good characterization of the maximum values. The maximum value is difficult to characterize perfectly, as a single outlier on any of the processes changes the maximum for a given iteration. Still, the value map of the same metric reconstructed from 128 clusters is very close to the original, and the important features needed to find the performance problem in this application (the dark diagonally moving lines and the growth of the point-to-point communication time with the process rank) are already clearly visible with 64 clusters.

Figure 4.14 compares the MPI time profile of the full 1300-iteration *PEPC* analysis report with a reconstruction from 128 clusters. Aggregate metric values for entire timesteps are in broad agreement, with up to 5% error for timestep 1292, and even the detailed call-path profiles show good agreement, with the selected `MPI_Waitany` call path in the penultimate timestep 1299 having only 9% error. The distribution of the selected call-path metric values over the 1024

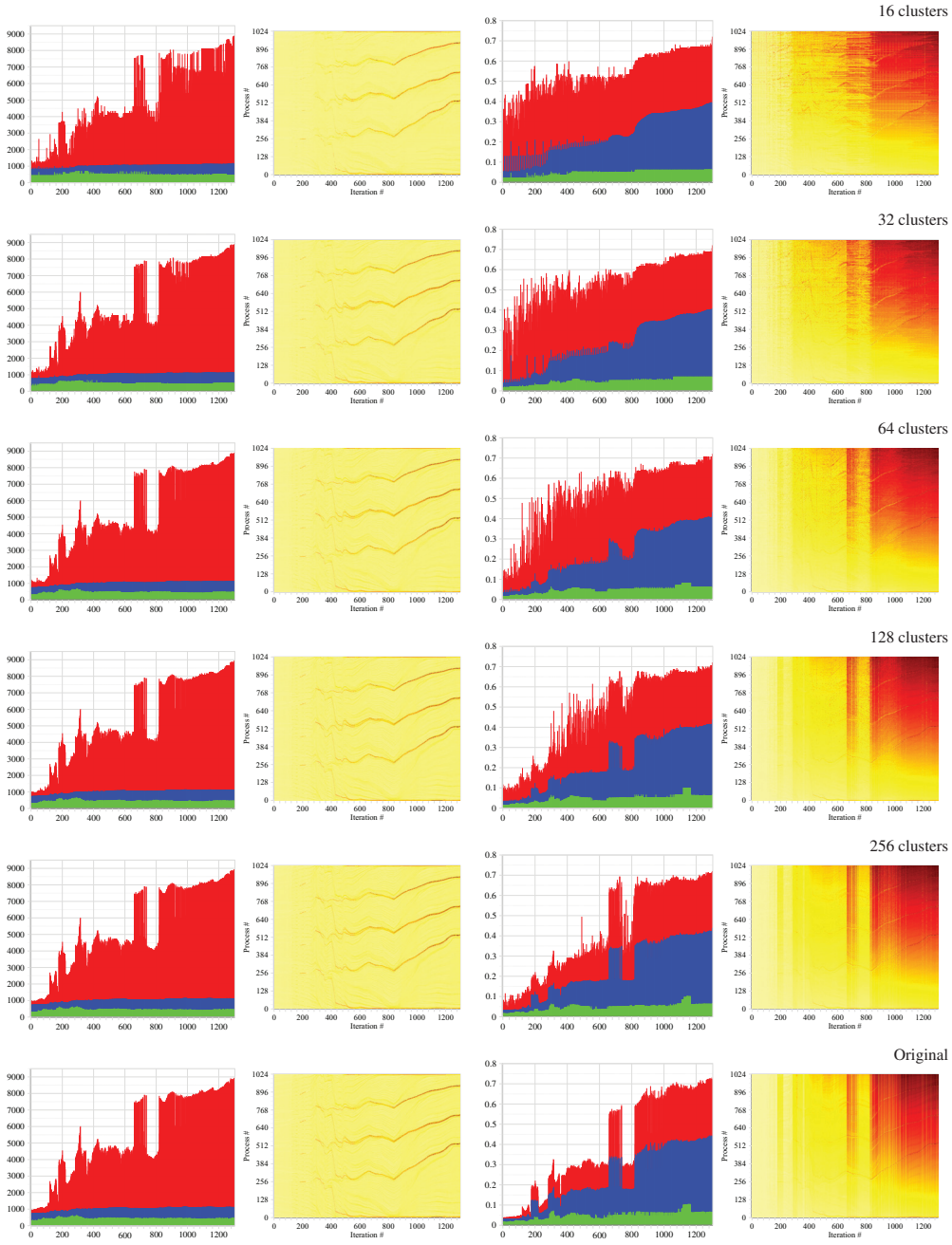


Figure 4.13: *Comparison of full-fidelity (bottom) and reconstructed iteration graphs and value maps of MPI point-to-point communication count (left columns) and time (right columns) for PEPC with different cluster counts.*

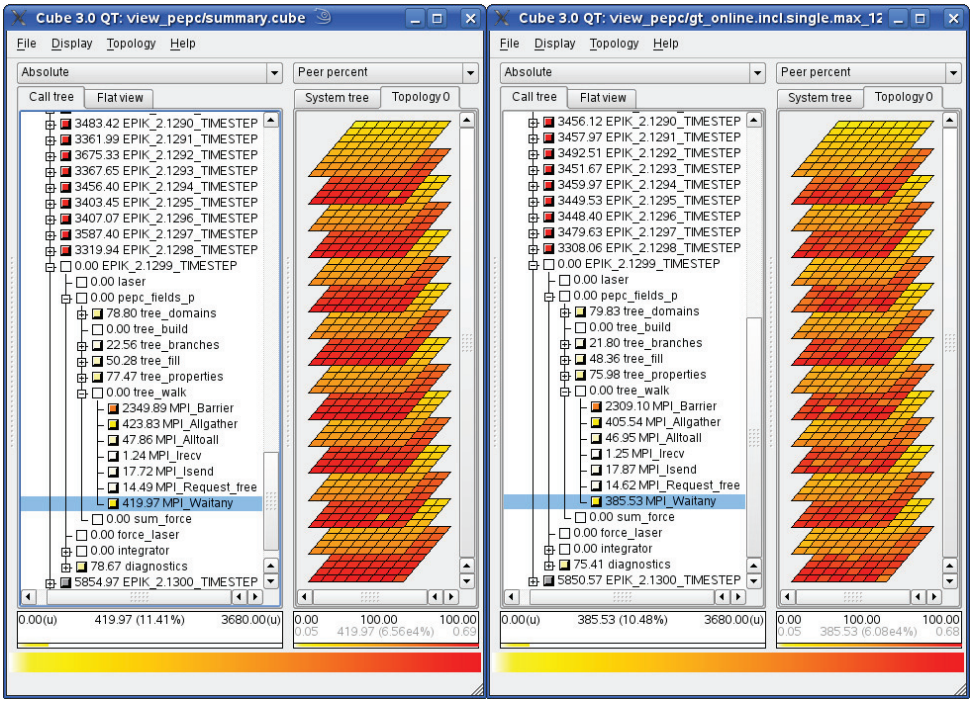


Figure 4.14: *Scalasca analysis reports of the full fidelity PEPC measurement (left) and that reconstructed from 128 clusters (right), showing the MPI time metric, and with a call path ending in* `MPI_Waitany` *selected in the* `tree_walk` *routine of the penultimate timestep.*

processes of the Blue Gene topology also shows reasonably good characterization, although significant differences for certain processes are evident.

At least 256 clusters therefore seem to be required to characterize all important features of *PEPC*, which means that we need less than 20% of the buffer space to store the call-path profile data at run-time, and still get sufficiently high quality data. This is especially important for *PEPC* on a Blue Gene machine with limited per-core memory: the developers naturally optimize their code using as much of the machine's memory as possible, leaving only a small amount for Scalasca measurement data.

4.5 Summary

A method for time-series call-path profiling is introduced and evaluated which uses incremental clustering of call trees with similar metric values for space-efficient storage. While using a distance function based on aggregate metric values allows efficient clustering decisions, accuracy of the compressed call-path profiles can be ensured by enforcing call-tree

equivalence based on visited call paths and heuristics that reduce the impact of extrema. An implementation is demonstrated which has acceptable processing time and memory overheads.

Whereas the call-tree and execution characteristics of the simplest applications such as *137.lu* lend themselves to accurate representation with as few as 16 clusters, more complex applications are found to need more, with 64 clusters being enough for most of them. For the *PEPC* application, however, at least 256 clusters are required for good time-series call-path characterization of its complicated execution behavior. 64 clusters therefore generally seems to be a reasonable default setting for a first measurement, as this value is usually high enough to get good compression quality and still has relatively low time and memory requirements. When considered insufficient, measurement can be repeated with a larger number of clusters. Still, one of the weaknesses of the compression technique is that it is hard to tell if the right number of clusters was used. When the cluster count is way too low, longer constant intervals appear in the time-based metrics and noise mostly disappears, which can be suspicious, but probably only to an experienced user. The best one can do to avoid this problem is to still use relatively high cluster counts, probably never less than 64.

Evaluation of the compression algorithm using the off-line prototype gave us much valuable insight and confidence to go on and implement the algorithm in the actual Scalasca measurement system, for which it was intended. The evaluation of those results is presented in Chapter 6, but before starting with this final evaluation, we present a novel measurement method in Chapter 5 we introduced to make low-overhead measurements possible even for applications where our standard instrumentation methods fail to deliver acceptable results. This new method sacrifices the capability of collecting exact data for achieving lower overhead, while still collecting all the necessary MPI communication data to achieve high-quality compression using the algorithm introduced in this chapter.

Chapter 5

Combining Sampling and PMPI Event Profiling

As we have seen earlier from our measurements in Figure 2.1 on page 21, the Scalasca measurement system that we build on in this thesis is capable of collecting low-overhead measurements of most of our test applications, requiring filtering in some cases to lower the overhead. Still, there are a few codes where it fails to deliver adequately low measurement dilation, resulting in executions so perturbed by the measurement itself that the collected data is close to useless. In this chapter, we introduce a new measurement methodology capable of overcoming these problems by selectively applying a lower overhead, less precise measurement method to the less important, computational part of the execution, while still collecting precise information about communication-related events. As the primary focus in performance profiling is on communication, having exact data about communication and statistical data about the computational part is just the right amount of information for meaningful analysis. This combination enables us to take high-fidelity, low-overhead measurements while still collecting enough information for the compression algorithm introduced in Chapter 4 to work properly. The compression algorithm relies heavily on the availability of the count-based communication metrics, which this technique can provide through the direct instrumentation of the communication events.

One can broadly distinguish between two performance measurement techniques: *sampling* and *direct instrumentation*. The first approach periodically samples the program counter as the program progresses to measure performance aspects statistically. From the program counter value, a profiler can easily derive the function the program was executing when the sample was initiated by an interrupt. We then estimate the fraction of the overall runtime spent in a given function by the fraction of samples that occur during its execution. In contrast, direct instrumentation (or *event-based* instrumentation), inserts hooks at function entry and exit points. This insertion can occur at levels ranging from the source code to the binary file or even the memory image [78]. These hooks allow the profiler to maintain a shadow stack at runtime. The stack stores the time at which a function is entered, which is subtracted from the time when the function returns. The accumulated differences precisely capture how much time was spent in each function. The Scalasca measurement system that this work is based on uses direct instrumentation as its measurement approach.

Many tools build a dynamic call tree of the application execution to attribute times to individual call paths, as introduced in Section 1.3. Such a call-path profiler tool can determine

the current call path in two different ways: It can maintain a shadow function stack, keeping track of the current call path by taking advantage of the deterministic characteristics of direct instrumentation; or it can use the technically more complex *stack unwinding*, sometimes also referred to as *stack walking* [9]. While stack unwinding is applicable to both direct instrumentation and sampling, it is typically only used in combination with sampling, as it is expensive, and the simpler and more efficient shadow stack-based method can be used in the direct instrumentation case. In the sampling case, keeping track of the current call path is not possible, as it does not provide a continuous stream of function entry and exit events, and any number of unknown changes to the current call path can be made between two sample interrupts without our knowledge.

Sampling and direct instrumentation both have advantages. We can easily control measurement dilation under sampling by adjusting the sampling frequency. However, it delivers an incomplete picture, potentially missing critical events or providing inaccurate estimates. Moreover, because the timer interrupt can occur at arbitrary program locations, sampling complicates accessing not only the current call path, but any details of the program state, such as the arguments of the current function. As a result, many tools resort to direct instrumentation to capture communication metrics such as message payload sizes. The MPI profiling interface [60], which leverages direct instrumentation through interposition wrappers, reflects this insight. However, direct instrumentation can result in excessive measurement dilation if applied indiscriminately, which quickly becomes apparent in C++ programs, which often have many short but frequently called class member functions. While heuristics can balance the amount of direct instrumentation with the need to cover all relevant program regions, they often require additional program runs.

Our approach combines the two methods in a unified framework that exploits the strengths of each. Specifically, we make the following contributions:

- The design of a call-path profiling technique that combines direct instrumentation of MPI routines with sampling of all other program regions;
- Modifications of the call-path profiler of the Scalasca toolset [28] to implement this technique;
- An inexpensive and portable enhancement of the classic stack-unwinding mechanism that can identify the call paths of frequently called communication routines without incurring excessive overheads;
- A study of the *DROPS* fluid-dynamics code written in C++ that demonstrates how our profiling technique sufficiently reduces measurement intrusion to take low-overhead measurements of a challenging real-world code and gain insight into its execution characteristics.

Overall, our approach captures accurate call-path profiles with a variety of communication metrics for a broad range of applications – including the growing class of C++ programs – without requiring cumbersome selective function instrumentation or expensive runtime filtering mechanisms.

The next section presents our call-path profiling approach and its integration into Scalasca. In Section 5.2, we conduct a detailed experimental comparison of our new method to Scalasca's

traditional profiling options, giving evidence of cases for which it was the only way to achieve acceptable overhead. Section 5.3 presents our study of the *DROPS* fluid-dynamics application, and finally Section 5.4 gives an evaluation of the compression algorithm introduced in Chapter 4 when applied to data collected using the hybrid call-path profiling method.

5.1 Hybrid call-path profiling

Call-path profiling requires two ingredients: (i) a mechanism to determine the call path at a given point in time and (ii) a method to decide when such a point has arrived. In our approach, we use stack unwinding for the former and combine sampling with direct instrumentation for the latter.

We conceptually divide the execution of each MPI process into two disjoint, alternating phases – execution outside and inside the MPI library. Outside the MPI library, we use sampling, which is independent of the frequency of routine entries and exits and, thus, provides better control of runtime overhead. Inside the MPI library, we use direct instrumentation, which greatly facilitates calculation of metrics based on parameters of MPI routines, such as the total number of bytes sent. We can easily intercept MPI calls with PMPI wrappers by relinking or dynamic loading, thus avoiding complex code transformations required for some direct instrumentation mechanisms.

Stack unwinding is central to our approach. Maintaining a shadow stack would require direct instrumentation of user code, but we only use direct instrumentation for MPI routines. In contrast, stack unwinding is essentially a stateless operation that can be applied at any time during program execution, and it allows the call path that leads to the invocation of a particular MPI routine to be determined whenever it occurs. Thus, stack unwinding is triggered by two different classes of events, either when a timer interrupt occurs during computation or when the program enters an MPI routine.

Another distinction concerns the attribution of the time spent in these two states. We calculate the time spent inside MPI based on entry and exit timestamps, while we assign the time spent outside MPI to individual call paths based on the samples that exhibit them. Our solution combines the advantages of both techniques: rich and reliable MPI performance information including messaging statistics and a low-overhead approximation of application performance elsewhere with small, frequently called functions having no impact on runtime dilation.

We integrated our solution into Scalasca, primarily targeting the reduction of measurement dilation during call-path profiling, particularly dilation occurring due to frequently called class-member functions in C++ applications. The previous version of the call-path profiler included in Scalasca relied exclusively on direct instrumentation to track the current call path and to determine the time spent in specific call paths, which made it vulnerable to excessive dilation caused by frequently called small functions.

Although source-code translators and binary instrumentation are also available, Scalasca most commonly employs automatic compiler instrumentation to insert hooks for direct instrumentation into user functions. Scalasca can dynamically filter out small but frequently called functions to lower the overhead of indiscriminate user-code instrumentation, as detailed

previously in section 2.2.1. These functions, which cause significant dilation but contribute little to the overall execution time, are placed on blacklists that prevent them from invoking the code that was added during instrumentation. However, this approach usually requires an extra run of the application under full instrumentation to identify the functions to place on the blacklist. Moreover, just making the filtering decision for the blacklisted functions takes some time, which can lead to significant overhead when dealing with functions that are called extremely frequently. Our new hybrid approach eliminates these costs.

Our approach generically builds on third-party libraries to perform the stack-unwinding procedure, which provides increased portability and flexibility. We focus primarily on an implementation that is based on libunwind v0.99 [63], an easy-to-use C library that supports a range of platforms and offers advanced features such as changing the instruction pointer, the stack pointer or different processor registers. The latter capability directly supports our stack unwinding optimizations. Even though we focus on the libunwind implementation and x86_64 platform here, our optimizations are independent of the platform or the unwinding library, and there is ongoing work to port our implementation to PowerPC (JUGENE) based on StackwalkerAPI v1.2 [17], a C++ library with stack unwinding functionality similar to that of libunwind.

Our approach faces two challenges. First, we must reduce stack unwinding overhead in the presence of frequent MPI calls. Second, we must integrate sampling-based and event-based time measurements within the same experiment. We discuss these challenges in the following subsections.

5.1.1 Fast call-path unwinding

Although we will see in Section 5.2 that our approach avoids excessive measurement dilation of user-code profiling through sampling, unwinding the call stack during every MPI call can still dilate measurements significantly if MPI calls are frequent. While we can easily adjust the sampling rate, the rate at which an application calls MPI functions is beyond our control. We therefore introduce several optimizations to lower the overhead of stack unwinding including non-trivial measures such as caching function start addresses and thunk stacks.

Unwinding only relevant MPI functions

In extreme cases, MPI call frequency exceeds the the sampling frequency of 100Hz that we use in our implementation by orders of magnitude. Many applications repeatedly invoke auxiliary MPI functions with insignificant execution times. For example, in the SPEC MPI2007 application *142.dmilc*, more than 99.5% of all MPI calls are to `MPI_Comm_rank`, while 68.4% in *147.l2wrf2* are to `MPI_Cart_shift`. We exploit Scalasca's ability to configure measurement of certain groups of MPI wrappers individually to turn off stack unwinding for a broad range of MPI functions that are not performance critical. However, we still count the occurrences of those calls where unwinding has been disabled with negligible overhead, which facilitates eliminating unnecessarily frequent ones.

Caching region identifiers

Using libunwind to determine the name of a function with a given start address incurs significant overhead. We also incur a high cost to map function names to region identifiers by which Scalasca uniquely identifies the functions internally. Thus, we implement a hash table that maps start addresses to their corresponding region identifiers, which eliminates the name lookup and string matching except for each function's first occurrence.

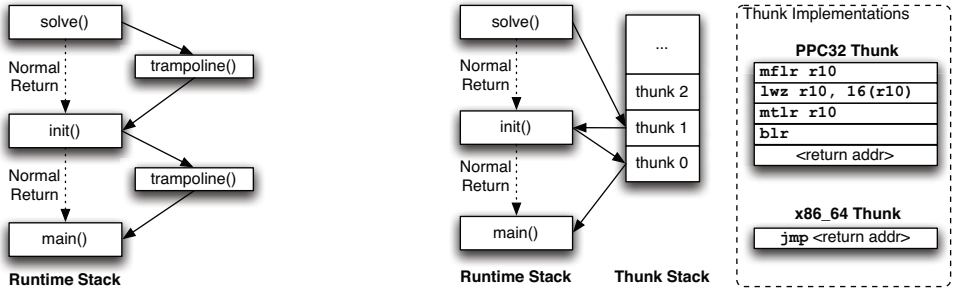
Caching start addresses

We must use another, even more expensive libunwind call to look up the start address of the function being executed based on the current value of the instruction pointer. Caching this information using a hash table is non-trivial because the timer interrupt can occur during execution of essentially any instruction, which would present a large set of keys for non-trivial applications. Fortunately, this problem only applies to the topmost stack frame. For all other frames (and the topmost stack frame when unwinding from MPI wrappers), the instruction pointer must refer to an instruction just after a function call site, corresponding to the return-pointer value of the function being called. Since the largest applications have only several thousand call sites, we can use a hash table to look up the function start address for most stack frames. We handle instruction addresses in the topmost stack frame (i.e., those the timer interrupts) with a separate, less efficient look-up structure, in which we check if the address falls in between the start and end address of a known function. Caching the start address is our most effective optimization, removing 90% of our unwinding overhead.

Light-weight thunk stacks

The optimizations that we have discussed so far either reduce the number of unwind operations or the time to perform actions related to a single stack frame. Our next optimization reduces the number of stack frames that we must examine to determine the full call path. Specifically, we detect when the current frame is the last element of a prefix of a previously unwound call path. Since we already know the prefix, we can avoid re-examining its stack frames. To mark prefix frames, we change each frame's return address as we unwind the stack. Based on these special return addresses, we can later identify marked frames and associate each with its corresponding prefix.

Changing bits in return addresses to mark frames could alter the program's control flow. Instead, we allocate a special region of contiguous memory to hold a *thunk stack*, shown in Figure 5.1(b). The thunk stack is composed of entries, or *thunks*, that mirror the state of the runtime stack when it was last unwound. Each time we walk over a stack frame during unwinding, we create a thunk in the stack that branches to that frame's return address. We then modify the original return address to point to the thunk. Since we know that instrumented return addresses will fall somewhere inside the thunk stack, we can identify them by comparing each return address to the thunk stack's location. Entries in the thunk stack are constant-size, so we can compute the depth of a particular prefix by subtracting the address of the bottom of the thunk stack from an instrumented return address.



(a) Trampoline function. (b) Thunk stack.

Figure 5.1: *Prefix optimization mechanisms.*

Other tools use different mechanisms to perform the same optimization. HPCToolkit uses a lightweight trampoline, as shown in Figure 5.1(a) [20]. Instead of pointing return addresses to a stack, the topmost function of a prefix is instrumented to return into a trampoline function. The trampoline performs bookkeeping for profiling and tracks the frame in the dynamic call tree to which it currently points. Upon return, the trampoline installs itself in the next lower stack frame so that that frame's function will also call the trampoline when it returns.

A thunk is much simpler than the HPCToolkit trampoline function. Figure 5.1(b) shows our thunk implementations. On x86_64, the 64-bit jump performed by the thunk is a single instruction, and on PowerPC it only requires four instructions. Further, our thunks do not modify the stack as the trampoline does. We thus avoid signal safety issues within our instrumentation code, as our sampling signal handler does not need to check whether it is within the trampoline to avoid conflicts. Our instrumentation is confined within unwind routines and isolated from our thunks.

This arrangement decouples our optimization from the particular stack unwinder used. The trampoline approach requires that trampoline code can interpret stack frames and insert itself in the proper location in the next frame. Our thunks do not perform stack surgery and, thus, can be implemented separately from the main stack unwinding logic, which makes our implementations less complicated than trampoline-based implementations. Stack unwinding APIs only must support writing to return address locations as we unwind the stack, a feature that both libunwind and StackwalkerAPI provide.

Finally, our approach avoids the problem of non-local function exits. In languages that implement exception handling, a routine may return into a routine other than its caller. This event causes control flow to skip the return that would install the trampoline in a lower frame. The trampoline approach thus must instrument all non-local exits to routines, which requires more complex code analysis [20]. Our approach avoids this analysis by simply placing a thunk at every level of the call stack. Thus, our scheme already instruments the frames in the prefix of any frame that a non-local exit skips, whereas a trampoline may be skipped entirely by a non-local exit.

Counting call-path visits

Our thunks support efficient counting of the number of times that a call path is visited. If a stack-walk step encounters a function in which we have installed a thunk, then this function has not returned since the previous unwind. Had it returned and been called again, its return address would not point to the thunk. This count is only an estimate, because we do not necessarily take a sample at every execution of a given call path, and we do not instrument non-local routine exits. However, this count is a guaranteed lower bound, as the function had to be entered and exited this number of times. Further, a unique property of our solution increases the accuracy of this lower bound. We perform stack unwinds not only during arbitrary timer interrupts but also inside every PMPI wrapper. Thus, we frequently unwind COM call paths (as introduced on page 22). In most cases, functions on these call paths always call at least one MPI function (directly or indirectly) every time they are executed, making sure that we do not miss any instances of those COM function calls. We might unwind from several MPI calls from within a single COM function call, but this does not cause a problem in the counting as the thunks indicate when we have not left a call path between unwind operations. These properties together mean that our visit count is likely to be exact for COM call paths. Overall, we provide lower bounds for the visit counts of arbitrary call paths, and exact visit counts for most COM call paths (those that call MPI every time they are called). HPCToolkit similarly tracks the visit count, but does most of the bookkeeping work inside its trampoline, whereas we do this work within our unwind calls.

5.1.2 Hybrid sampling methodology

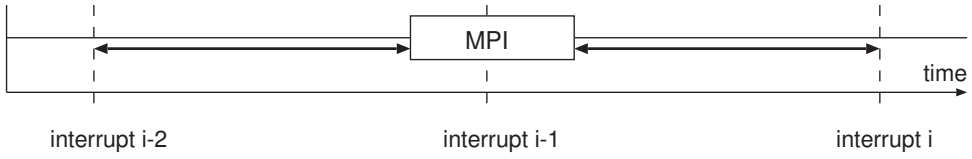
Our hybrid profiling approach combines two profoundly different measurement modes. This combination requires that we prevent undesirable interference between their underlying mechanisms. Further, we must integrate their measurements in a consistent and statistically sound manner.

Timer interrupts inside MPI

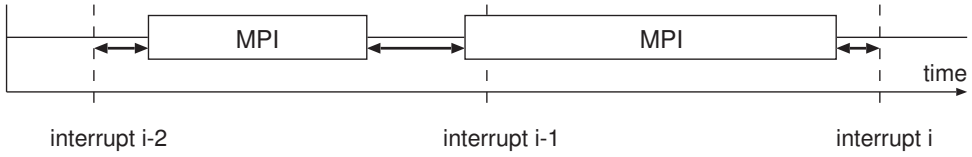
A possible interference between the two modes arises when timer interrupts occur during MPI calls. Although at first glance it may seem reasonable just to pause the timer whenever control is transferred to MPI, the high frequency of MPI calls makes this approach extremely expensive. Thus, we simply ignore interrupts that occur inside MPI functions. We explain below how we correctly account for ignored interrupts and how we avoid inaccuracies due to MPI calls, which are not part of the sampled population.

MPI calls inside sample intervals

While we directly measure the time spent inside MPI, we only statistically measure the time spent outside MPI (which we refer to as *computation* in the following). We must separate these two measurement realms so that we represent each of them as accurately as possible



(a) Effective sample interval length greater than normal.



(b) Effective sample interval length smaller than normal.

Figure 5.2: *The impact of MPI calls on the effective sample interval length.*

while maintaining the invariant that their sum equals the directly measured overall execution time.

Classic sampling methodology estimates the *Execution time* t_c consumed by a certain call path c as:

$$t_c = \frac{n_c}{n} * t$$

with t being the total execution time, n being the total number of samples, and n_c being the number of samples exhibiting call path c. We could calculate the execution time within a computational call path either by including the ignored samples into n, as if we were sampling everything, or by rescaling the equation variables as if the entire execution consisted only of computation. Either choice could result in inconsistent timings due to the representation of overall MPI time in two different ways, directly via explicit wall-clock readings and indirectly via the number of interrupts that occur inside MPI.

The frequency of MPI calls makes these solutions inadequate. Our experiments confirm that the two representations correspond well, as long as applications alternate between large contiguous phases of computation and communication. However, there are slight deviations when the frequency of MPI calls rises. Further, time assignments may be biased depending on when a call path primarily executes at runtime, because the MPI call frequency may differ not only between applications, but also between different phases of the same application.

Classic sampling methodology assumes evenly distributed samples across the execution time with intervals between consecutive samples having about the same length. Our hybrid approach violates this assumption for two different yet related reasons. First, we drop timer interrupts that occur inside MPI calls. Second, the interval between two valid interrupts may include MPI calls, which influence the effective length of the interval. Depending on the duration of these MPI calls, the effective length of this interval can be shorter or longer than the normal timer interval. As illustrated in Figure 5.2, we define the *effective sample interval length* of a sample taken at interrupt i as the sum of the computation intervals since the last interrupt outside MPI ($i-2$ in the figure). We can calculate this length as the distance between i and $i-2$ minus the intervening time spent in MPI as determined by our direct

measurements. Depending on the extent of MPI execution, the effective length can be greater than (Figure 5.2(a)) or less than (Figure 5.2(b)) the regular timer interval length.

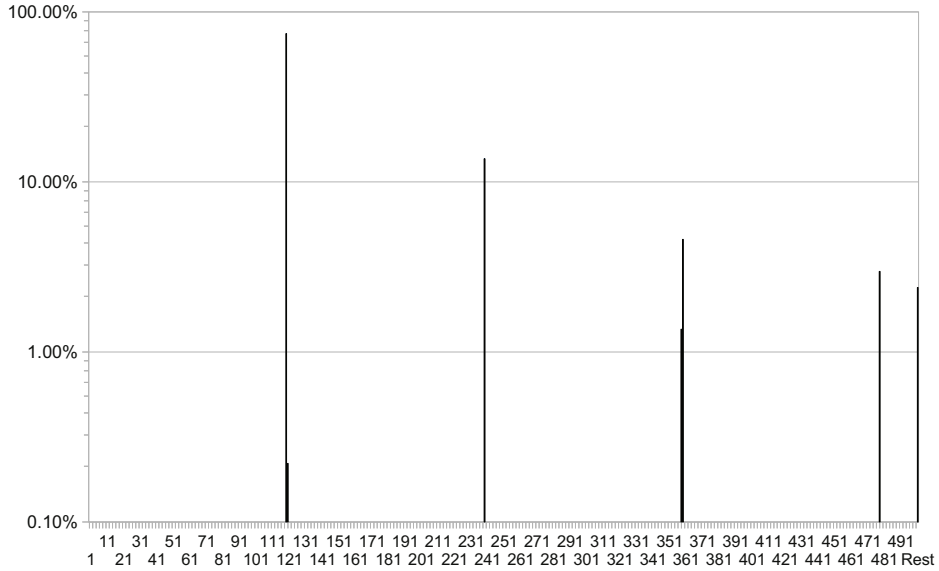
Instead of excluding the MPI time at a global level by rescaling all measurements by the same factor, we use a more flexible solution that locally rescales the time attributed to a sample. This approach accounts for local fluctuations of the sampling frequency caused by MPI calls. Our solution assigns a weight that corresponds to the length of the effective sample interval to each sample taken outside MPI. Thus, we use a variable sampling frequency that assigns more weight to samples that are taken in regions with a lower *effective sampling frequency* and less weight to call paths in regions with a higher effective frequency. At the same time, this approach ensures that effective sample intervals and MPI times add up to the overall execution time, preserving consistency as desired.

Figure 5.3 presents an example where we require this approach, the SPEC MPI2007 application *128.GAPgeofem*. The bins in the histogram in Figure 5.3(a) represent the sample interval before we subtract the time spent in MPI. 75% of the intervals have normal length, 14% are twice as long, 6% are three times as long, and the remaining 5% are more than three times as long. Figure 5.3(b) shows the histogram after we subtract the MPI times, resulting in the effective sample interval length. 34% of the intervals have exactly the normal length, which means that they did not include any MPI execution. Another 42% of the intervals have a length between zero and the normal timer interval length. These intervals typically have the normal length (they are in the first peak at bucket 120) in Figure 5.3(a), but their lengths are reduced in Figure 5.3(b), as they include some time in MPI calls.

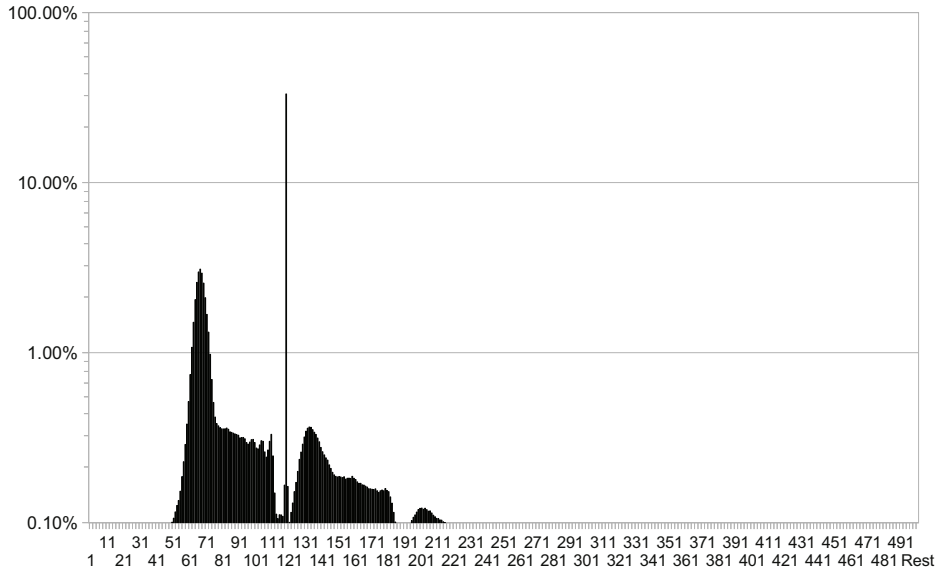
When looking at Fig 5.3(b), we see a local maximum around bucket 70. As the normal interval length falls into bucket 120, this indicates that the typical amount of time spent in MPI in those sample intervals where MPI calls are present is slightly less than half of the sample interval for this application. We also see a dip to the left of the normal interval length, between buckets 110 and 120, as any given interval rarely includes less than 0.001s of MPI execution. Intervals that comprise the second peak of Figure 5.3(a) typically appear to the right of the normal interval length in Figure 5.3(b). These intervals, which must have included MPI execution (otherwise they would be normal length intervals), are now distributed similarly to the samples from the first peak, but at a different scale (on the logarithmic y-axis). The similarity in the distributions is due to the fact that both sets of values were generated based on the same MPI time distributions, just subtracting these MPI distributions from one or two sample interval lengths, respectively. The histogram shows that even though we allow variable and theoretically unbounded sample interval lengths, most of the interval lengths are very close to the normal timer interval. Having many arbitrarily long sample intervals would not invalidate our method, but would slow down its convergence, which means that we would need longer executions to get high quality statistical data.

5.1.3 Implementation challenges

While the basic ideas behind the hybrid measurement method may be simple, implementing the prototype on a real-life system offered significant challenges. A selection of the most interesting challenges is discussed in this section.



(a) Sample interval length before subtracting MPI time.



(b) Sample interval length after subtracting MPI time.

Figure 5.3: *Logarithmic histograms of sample interval lengths with 0.0001s resolution buckets on the* x*-axis for 128.GAPgeofem; the last bucket contains all intervals that do not fit into the first 500 buckets.*

Compiler optimizations

Unwinding the call stack on the x86_64 architecture requires great care. Some compiler optimizations such as tail calls or even hand-written assembly code may confuse libunwind, leading to incorrect return values [87]. Especially when using thunks, bogus return values from the unwinder will almost certainly lead to a fatal error. To avoid such errors, we validate libunwind return values by ensuring that the preceding instruction was a call to the corresponding function. If it is not, we simply do not install the thunk. StackwalkerAPI did not exhibit this problem.

Signal safety

Each function called from our interrupt handler must be signal safe. However, we require that our signal handler has access to the full Scalasca measurement infrastructure, which is not signal safe. We solve this problem through guards to every entry point of the Scalasca library. The guards check a single atomic flag (`volatile sig_atomic_t`) and immediately return if it is set. After the guards, the flag is set, and it is unset at Scalasca's exit points, which eliminates race conditions within Scalasca.

5.2 Experimental evaluation

We investigate the benefits of our techniques using the same set of SPEC MPI2007 v2.0 benchmark suite [83] applications we used for evaluating our methods in Chapters 3 and 4. We also use the *DROPS* application for evaluating our methods, since *126.lammps* is the only C++ application in SPEC MPI2007, and complex C++ codes are typical use cases where these techniques can be beneficial. The *DROPS* software package [38, 19, 36] is an object-oriented computational fluid dynamics framework that solves the problem of efficiently simulating two-phase incompressible flows. In our test case it is simulating a rising droplet in a fluid. *DROPS* is implemented in C++, and incurs prohibitive measurement overheads under direct instrumentation [48].

The measurements are taken on JUROPA using similar configurations as in Chapter 2. We compare two Scalasca measurement sets. The first set uses Scalasca's existing capabilities based on direct instrumentation, previously introduced in section 2.2.1. The second set uses our new hybrid sampling-based approach.

5.2.1 Direct instrumentation

The Scalasca instrumenter, when used as a prefix to the usual compile and link commands, configures the compiler to instrument the entry and exits from user-level source routines. Although details vary by compiler, almost all contemporary compilers offer such a capability. By observing these entry and exit events for routines, the Scalasca runtime library can track the current call stack of instrumented routines as they execute and update its call-path profile

accordingly on each process. As MPI events occur, delivered by the PMPI wrapper library, they naturally augment the call paths in the profile.

Figure 5.4 gives an overview of the runtime dilation observed at different instrumentation levels for each application. The green and red lines at 5% and 15% are the thresholds we used for categorizing overheads as acceptable, borderline or excessive. Of course these are arbitrary thresholds, but still realistic and based on common sense. The leftmost (red) bars in Figure 5.4 show that five of the applications have less than 5% measurement overhead, which is sufficiently low for most uses. While *143.dleslie* and *145.lGemsFDTD* (12%) are borderline, *147.l2wrf2* (19%), *129.tera_tf* (31%), *142.dmilc* (50%), *122.tachyon* (237%) have excessive overhead. Finally, the 9394% overhead for *DROPS* confirms that plain direct instrumentation is not capable of providing meaningful, reliable results in certain more challenging cases.

When we instrument and measure every user-level source routine, high dilation can occur, particularly for small/short computational routines that execute frequently. We can reduce this overhead significantly by filtering such routines with the filtering approach discussed in Section 2.2.1. Filtering means that we lose information about these routines, but they are typically less important routines where very little time is spent, such as overloaded relational or array indexing operators.

Filtering reduces overhead in every case, as the middle (green) set of bars in Figure 5.4 shows. The overhead reduction in the case of *125.RAxML*, *142.dmilc* and *145.lGemsFDTD* are substantial and result in acceptable overheads. In the cases of *129.tera_tf* and *147.l2wrf2* filtering proved to be useful, but less effective. Although the *122.tachyon* overhead is down to 47%, it remains a concern: with all overhead due to a single routine, static filtering could be employed to avoid instrumenting it entirely. While static filtering is a viable option, we generally consider it a more involved method for advanced users. *DROPS* measurement also benefits from a comprehensive filter (listing more than a thousand routines), yet the dilation of 227% may still result in undesirable distortion. We examine this issue in detail in Section 5.3. Overall, instrumentation of user-level source routines proves straightforward and effective for most but not all applications and is sometimes inconvenient. Ideally, we would like to have a measurement method that provides all relevant information with low overhead, but without any complicated manual configuration or the need to run the same experiment several times.

5.2.2 Hybrid sampling

In this section, we evaluate the new hybrid sampling-based measurement method, and show that it is a useful alternative to direct instrumentation, providing consistently low overhead already on the first run, without a need for filtering or any other complicated manual measurement configuration.

An important drawback of not using direct instrumentation is that we cannot simply update the current call path at every event by adding or removing one call from the previous call path. If we want to build a dynamic call tree, we have to do expensive stack unwinding from every event, both from signal handlers and PMPI wrapper functions. As we control the frequency of samples but have no control over the frequency of MPI calls, most of the

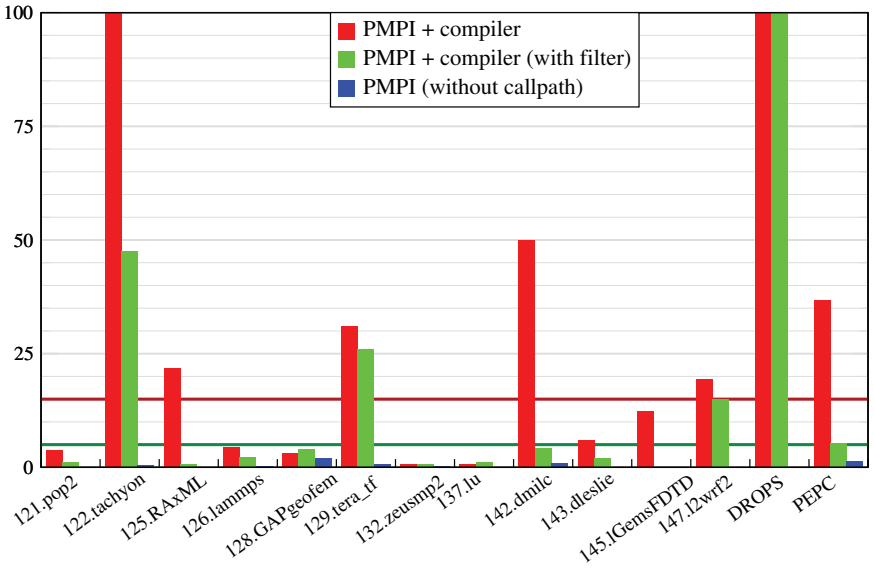


Figure 5.4: *Measurement dilation percentage for executions with PMPI wrapper profiling combined with full, filtered or no compiler instrumentation.*

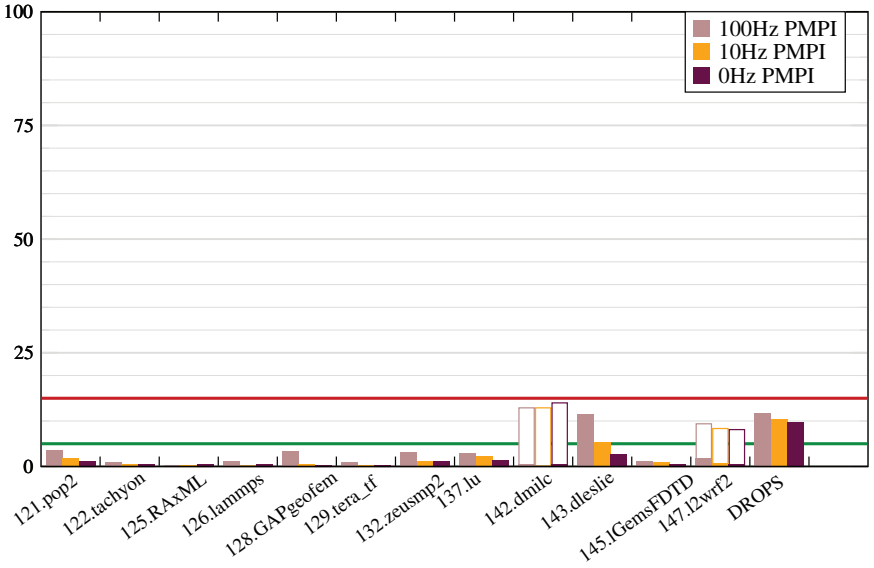


Figure 5.5: *Measurement dilation percentage for executions using the hybrid sampling method at different sampling frequencies.*

unwinding overhead often results from unwinding from PMPI wrappers, so it is also important to evaluate the measurement overhead in those cases where we turn sampling off completely and do stack unwinding from PMPI wrappers only. In these cases, the resulting call-path profile roughly corresponds to that produced by the filtering scenario that excludes all user-level source routines not on call paths to MPI operations. We expect small differences, for example when the compiler changes inlining decisions and from routines that the compiler does not instrument (such as Fortran interfaces to MPI library routines).

The rightmost (dark brown) set of bars in Figure 5.5 shows the measurement dilation when unwinding from PMPI wrappers only with no sampling. For most applications, this overhead is under 2%. Exceptions are *143.dleslie* (3%), *147.l2wrf2* (8%), *DROPS* (10%) and *142.dmilc* (14%). The *PEPC* application is not used for evaluating this technique as we measured it on the JUGENE system, where hybrid sampling is not yet implemented.

147.l2wrf2 and *142.dmilc* incur more overhead primarily because they frequently invoke certain MPI routines that we do not really need to profile. The Scalasca measurement library defines groups of MPI routines as shown in Table A.1 on page 138. In the hybrid sampling-based measurement mode we can use these MPI groups to selectively disable unwinding from these MPI calls while still collecting all other data about them. Although disabling these calls would be unnecessary when using plain direct instrumentation, the additional cost of frequent stack unwinding from such routines is prohibitive. Disabling unwinding from the `CG_MISC` group containing `MPI_Comm_rank` for *142.dmilc* and this group plus the `TOPO` group containing `MPI_Cart_shift` for *147.l2wrf2* reduces measurement overheads to under 1%. After noting that these functions are called very often, it is safe to disable unwinding for them, as they generally do not contribute much to the understanding of the overall execution characteristics. Also, we are still counting their occurrences and measuring time in them, we just do not do the stack unwinding from them, meaning that their exact position in the call tree will be lost in subsequent measurements. This is a tradeoff that we can safely make in order to ensure low-overhead, high-fidelity measurements. Figure 5.5 shows the best results with solid bars, and the two cases without this optimization with outlined white bars.

with an interval timer configured to deliver interrupts at a specified rate to each process, we can use stack unwinding from these signals to augment the call-path profile with non-MPI call paths approximating the profiles produced by (unfiltered) compiler instrumentation. This approach includes additional call paths since signals occur during execution of uninstrumented routines (such as those in system libraries). The leftmost (light brown) and central (orange) sets of bars in Figure 5.5 show measurement dilation when using one-hundredth of a second (100Hz) and one-tenth of a second (10Hz) timer interrupts, respectively. We find slightly higher measurement overhead as expected, but it remains below 4% for all SPEC MPI applications but *143.dleslie*, for which it rises to 12% at 100Hz. Overheads for *DROPS* are relatively high when compared to the other codes (around 12% at 100Hz), but much lower than in the direct instrumentation case (which was over 100% even with filtering).

We measured the time to perform stack unwinding from PMPI and non-PMPI events for each application, as Figure 5.6 shows for 10Hz and 100Hz. It is important to note that these overhead numbers are based on the reported time spent in different unwinding operations, and are not directly comparable to overall overhead numbers based on the ratio of instrumented

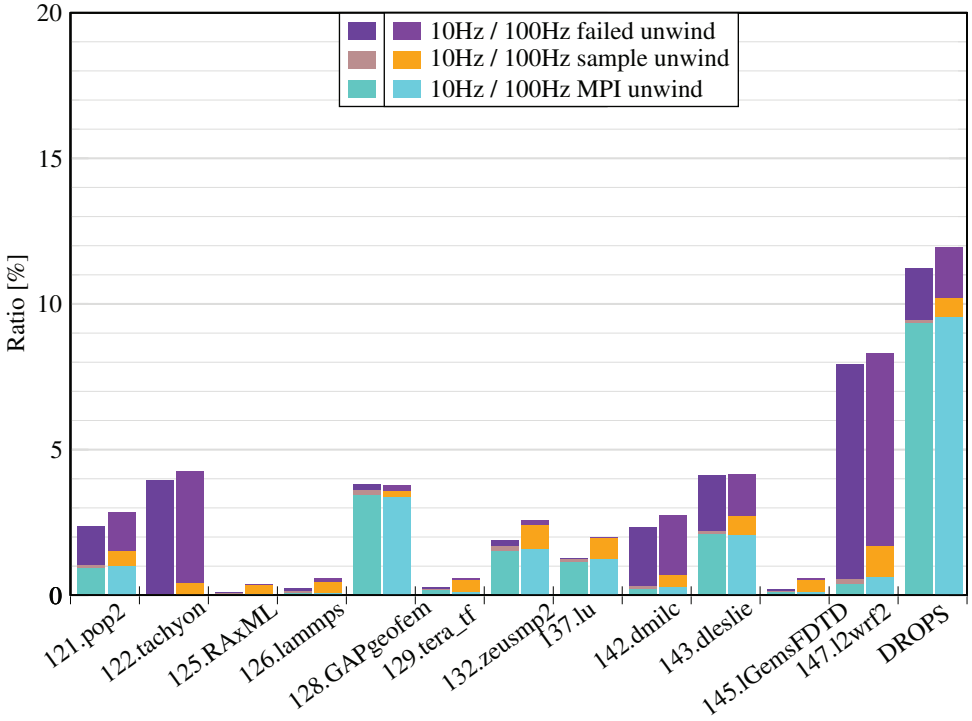


Figure 5.6: *Measurement distortion due to call-stack unwinding of MPI and non-MPI events including impact of unsuccessful unwinding.*

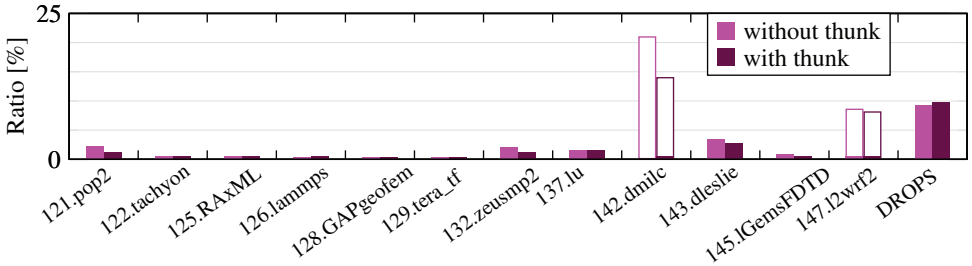


Figure 5.7: *Comparison of unwinding overhead percentage with and without the thunk optimization.*

and uninstrumented execution times. The PMPI unwind cost (lower/cyan bars) dominates in most cases, equivalent to over 9% of the execution time for *DROPS*, 3% for *128.GAPgeofem* and 2% for *143.dleslie*. Non-MPI unwind costs (middle/orange bars) are negligible at 10Hz and only close to 1% at 100Hz for *147.l2wrf2*, *132.zeusmp2*, *137.lu* and *143.dleslie*.

Stack unwinding from PMPI or non-PMPI events sometimes fails, which we categorize as "failed unwinds" in the Scalasca profiles. These failures typically occur within system libraries, hand-optimized assembly code in mathematical libraries, and the MPI library. The upper/purple bars in Figure 5.6 show that the proportion of time spent in regions where unwinding failed is generally low and roughly consistent regardless of the sampling frequency. For seven of the SPEC MPI codes call-stack unwind failures correspond to under 0.2% of execution time, with the worst cases being *122.tachyon* (3.8%) and *147.l2wrf2* (6.6%). For *DROPS* it is a moderate 1.7%.

We show libunwind measurement overheads with and without our thunk optimization in Figure 5.7. *142.dmilc* shows significant improvement, however, only when we include unwinding from unimportant MPI routines. Although the thunk does not provide a major reduction in overhead, at least on JUROPA, it still helps to provide more accurate call path visit counts.

Overall, our new approach provides reliable and convenient comprehensive call-path profiling of PMPI and user-level routines. We observe negligible measurement overheads (well under 5%) for all test applications except *143.dleslie* and *DROPS* on JUROPA. Those two worst-cases show moderate, 12% overhead, which is still acceptable, especially where the overhead from direct instrumentation would be significantly higher.

5.3 C++ application example: DROPS

As already seen in Figure 5.5, the DROPS application shows excessive overhead when using direct instrumentation (9394% without filtering, down to 227% when using aggressive filtering). This makes it a suitable candidate for evaluating the hybrid sampling measurement method. As we achieved significant reduction of measurement dilation using the hybrid sampling technique — down below 15% as seen in Figure 5.5 — it is important to evaluate if the decreased measurement dilation is only a quantitative difference or if it provides qualitatively different insight. We compare two key metrics, *Exclusive execution time* and *MPI collective communication time* in the following sections.

5.3.1 Execution time metric

Figures 5.8 and 5.9 show views of the Scalasca summary analysis reports of *DROPS* executions on JUROPA with 512 processes, one using direct instrumentation and the other combining PMPI measurements with 100Hz call-path sampling. Metrics listed in the leftmost panel apply to the entire measurement (e.g., time aggregated for all processes), which can be refined to selected call paths shown as a tree in the middle panel. The *Exclusive execution time* metric is selected here, which is also known as *Computation time*, as it contains the time spent in

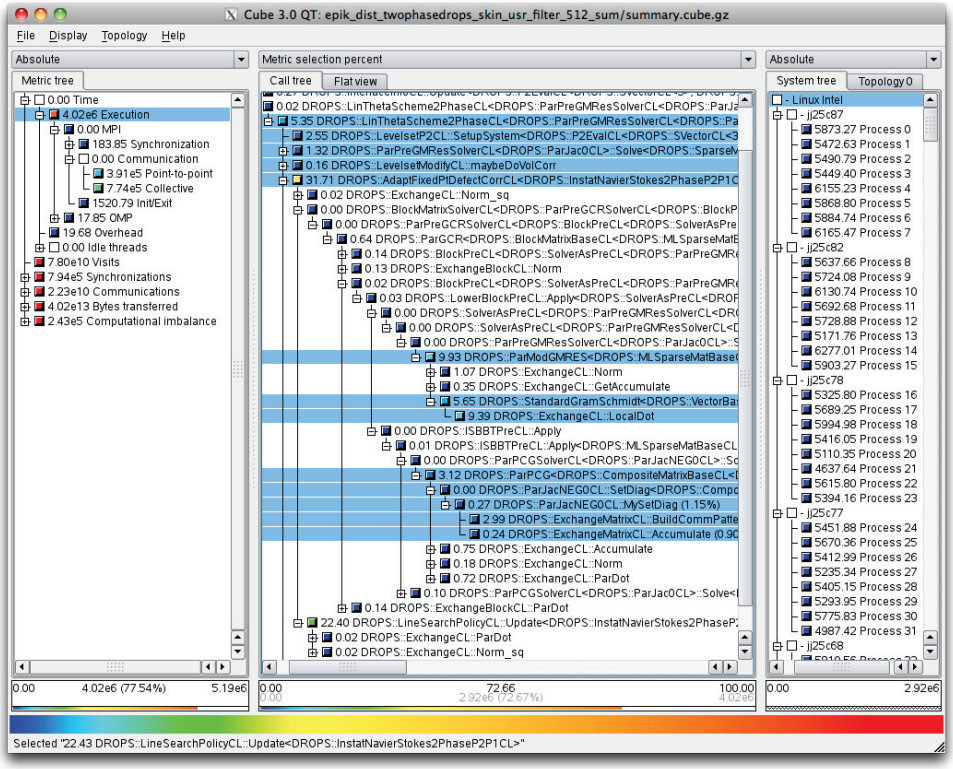


Figure 5.8: *Scalasca analysis report from a compiler-instrumented measurement of DROPS with 512 processes, highlighting call paths which are the hot spots in the Execution time metric.*

computation, but excludes the time spent in communication. The five most important call path groups are highlighted for comparison in both screenshots. We are not comparing individual call paths here as those are not directly comparable due to instrumentation differences like filtering and inlining. What we highlight are in all cases the equivalent call paths from both measurements.

First of all, it is important to note that the absolute values are often quite different, as the values from the direct instrumentation measurement are typically highly dilated, here by over 100% for the entire execution compared to 12% with hybrid sampling. Here we are concentrating on the ratio of time spent in each key call path group against the overall value of the metric. The most important finding is that the dilation affects the call paths in a non-uniform way, causing certain call paths to be shown as much more important than they really are. The direct instrumentation measurement shows the first highlighted group having 41.1% of the overall execution time, making it the most prominent in the measurement, while it only has 28.8% in the hybrid sampling measurement, which has significantly less dilation. Conversely, the hybrid sampling measurement shows the second highlighted call path group having about the

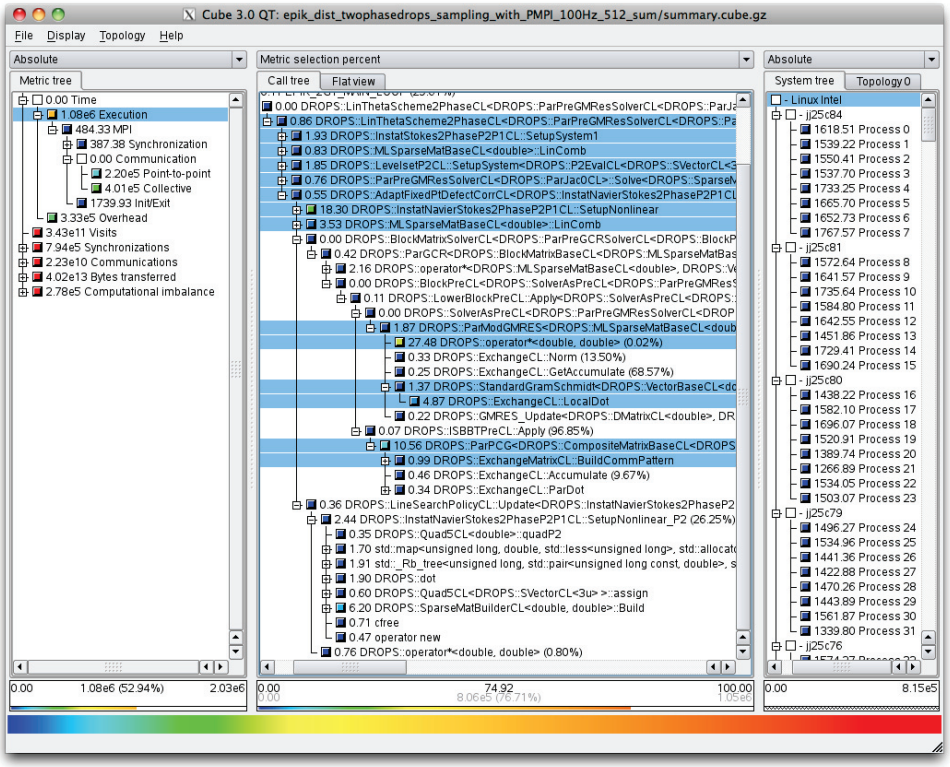


Figure 5.9: *Scalasca analysis report from a hybrid sampling-based measurement of DROPS with 512 processes, highlighting call paths which are the hot spots in the Execution time metric.*

same ratio, 30% of the overall execution time, while the same call paths account for only 9.9% of the overall execution time in the direct instrumentation measurement.

5.3.2 MPI collective communication time metric

After comparing computation times on the most significant call paths, the logical next step is to also compare measured times on the most important communication call paths. While there is a broad match in the comparison of the ratio of time spent in *Point-to-point* vs. *Collective communication time* (with the respective dilations applied), going down to the level of individual call paths shows that the heavily dilated direct instrumentation measurement is misleading in this case as well. In particular, the direct instrumentation-based measurement highlights the `MPI_Allreduce` inside the `ParTimer::GetMaxTime` call path as the most significant one, having 43.7% of the overall *MPI Collective communication time*, while it only has 4.5% in the hybrid sampling case, which in turn highlights the `MPI_Allreduce` inside `DROPS::StandardGramSchmidt` as the most important, with 55.3% of the *MPI Collective communication time* rather than 16.8% in the direct instrumentation case.

Correctly identifying the call paths where most of the communication time is spent is crucial in the process of performance analysis. It often reveals important workload imbalances or other factors that lead to waiting times in the communication. Again, the conclusions drawn from the two measurements would be very different on where to look for the most significant performance problems.

5.3.3 Deciding the question

In both of the previous examples we have two different measurement methods reporting largely different results on what the most important call paths are for directing further analysis. While the hybrid sampling-based measurement is likely to be more accurate, as it has significantly lower dilation, the results are not conclusive. To decide the question, we executed a third kind of measurement, using the MPI wrappers only and no direct instrumentation or sampling for the computation code. Instead, we inserted a minimal amount of source-code instrumentation to annotate the hot spots already identified, just enough to deliver the data necessary to decide this question. This measurement is aggressively tuned for minimal overhead, taking (and interpreting) such measurements is generally considered too advanced and labor-intensive for average users. It involves carefully placing manual source-code instrumentation at certain call-path depths to clearly differentiate the MPI calls we are interested in from the others, while keeping the measurement dilation as low as possible. Using this method, we achieved measurement dilation as low as 4.6%, which might still sound relatively high, but it is still much lower than either of the other measurement types could reach.

Figures 5.10-5.11 compare the measurement results from the three different measurement techniques. They compare the absolute values reported by the different measurement methods for the computation and the collective communication hot spots. The hot spots are named in the graphs after important, representative function names on their call paths. As we consider the measurement based on MPI wrappers and source-code instrumentation only as the most accurate, the more similar the bars representing the other two measurement methods are to the bar on the right hand side of each figure, the more accurately they represent reality. In Figure 5.10 the results from the hybrid sampling-based method matches nearly exactly those from the special low-overhead measurement, while the values reported by the direct instrumentation measurement are clearly heavily dilated, and not even close to reality. In Figure 5.11 the match is again much better in the hybrid sampling case.

Comparing relative values, to see how close the match is between the proportions derived from each measurement in Figures 5.12-5.13), we find again that the hybrid sampling measurement is in good accordance with the purely MPI wrapper-based measurement, while direct instrumentation was dilated unproportionately, leading to less reliable values. This comparison if perhaps even more important than the comparison of the absolute values, as it is generally the relative weight of a given call path from the overall execution time that we evaluate when looking for candidates for further analysis.

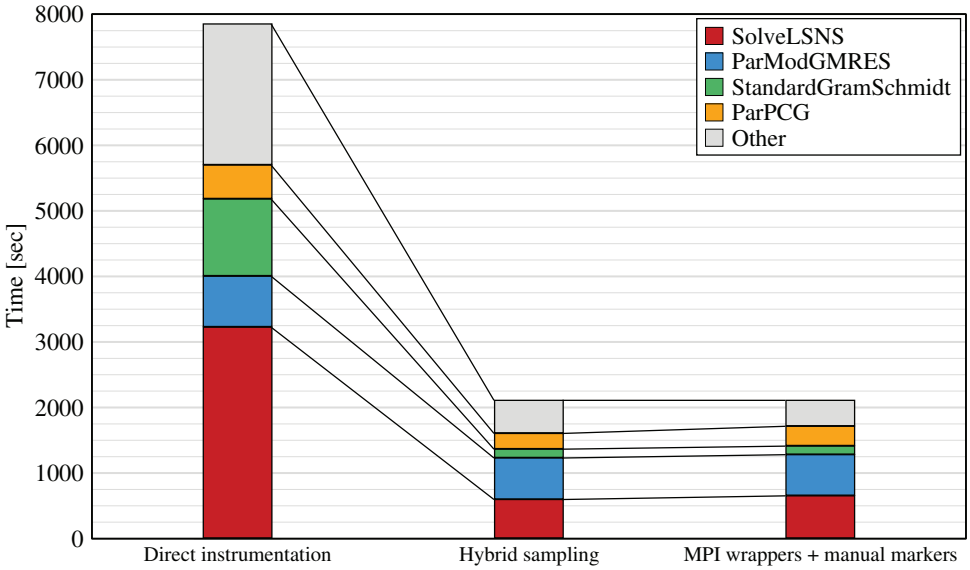


Figure 5.10: *Comparison of absolute time reported for the computation hot spots by direct instrumentation, hybrid sampling and pure MPI wrapper-based measurement.*

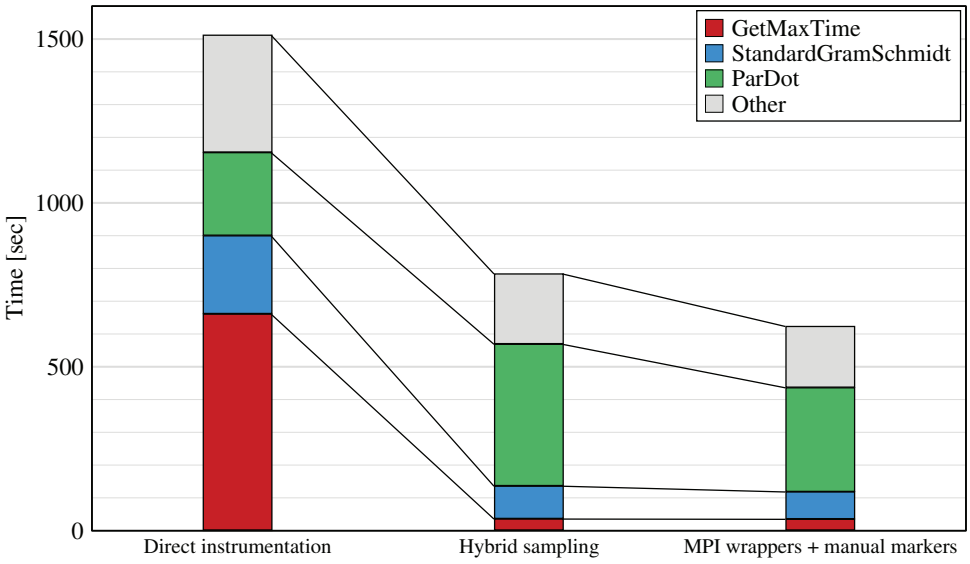


Figure 5.11: *Comparison of absolute time reported for the MPI collective hot spots by direct instrumentation, hybrid sampling and pure MPI wrapper-based measurement.*

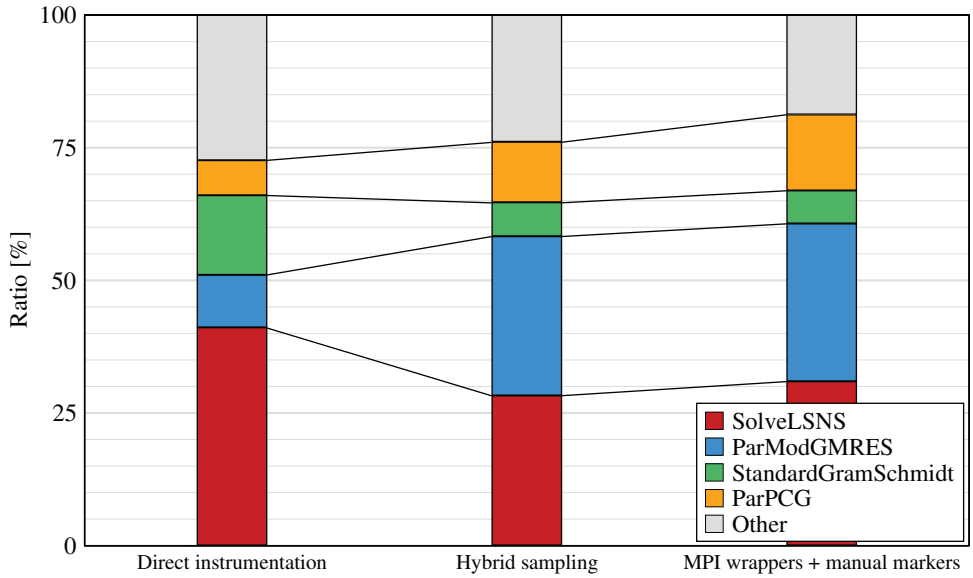


Figure 5.12: *Comparison of proportion of time reported for the computation hot spots by direct instrumentation, hybrid sampling and pure MPI wrapper-based measurement.*

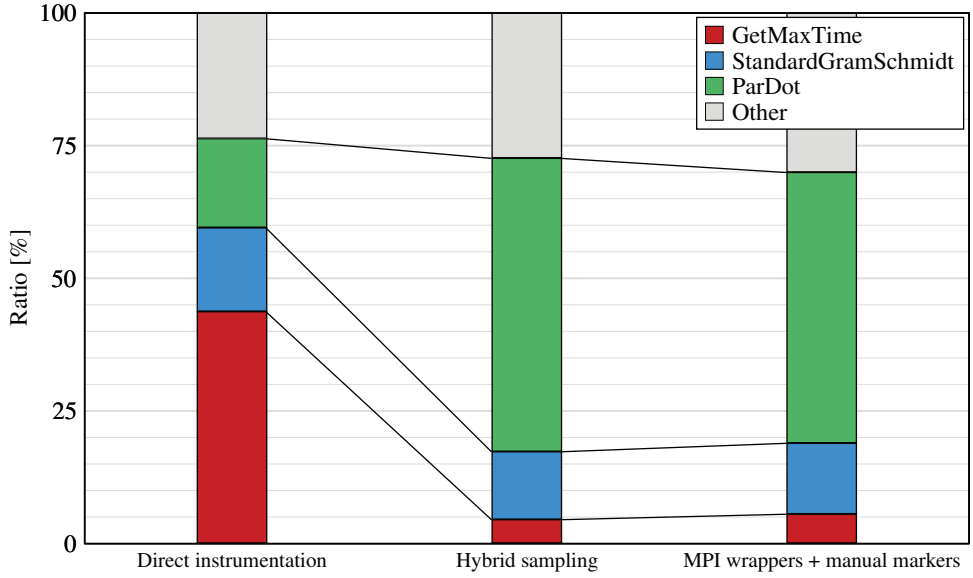


Figure 5.13: *Comparison of proportion of time reported for the MPI collective hot spots by direct instrumentation, hybrid sampling and pure MPI wrapper-based measurement.*

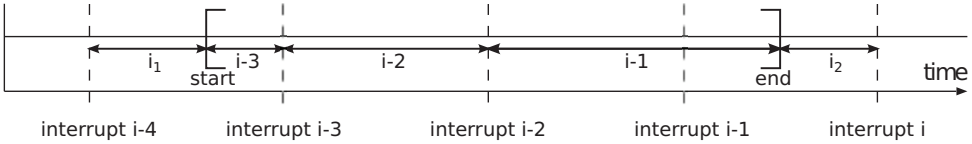


Figure 5.14: *The impact of OpenMP and marked regions on the effective sample interval lengths.*

5.3.4 Conclusion

While the two measurement approaches generate similar call trees and it is possible to match the call paths of the measurements to each other, it is often not a 1:1 match, as inlining and filtering make the call trees different in the two measurements. In particular, the aggressive filtering required by the direct instrumentation method to make a measurement feasible at all causes large parts of the call tree to disappear completely, limiting the insight gained into where most of the *Execution time* is spent. Moreover, as dilation affects different call paths in a non-uniform way, the heavily dilated direct instrumentation measurement leads us astray in both the *Exclusive execution time* and *MPI collective communication time* metrics, identifying the wrong call path as the most important one in both cases. This makes the hybrid sampling measurement superior in both the extent of details provided in the call tree and even more importantly the quality and reliability of the data presented to the user.

5.4 Compression of hybrid time-series profiles

When designing the hybrid sampling-based measurement method, one of the main design goals was compatibility with our existing methods. In particular, it was our goal to be able to use the iteration instrumentation and clustering-based compression capabilities on data collected using the new method. The most important feature enabling this capability is that we collect full information about MPI communication events, complete with all the relevant arguments such as number of bytes transferred. In this way, we have access to the full range of metric values required to get good representations of the iterations and compress them accordingly. In the following section we review the particular opportunities and challenges encountered during the process of bringing these features together.

5.4.1 Exact measurement of iteration and marked region times

The source-code instrumentation applied for marking source-code regions and iterations is a form of direct instrumentation, but slightly different from the usage of the PMPI interface. As our general philosophy for measuring time in hybrid sampling measurements is to use all available information for providing the most exact results possible, we wanted to take advantage of the fact that we have access to exact time stamps at the beginning and end of iterations. This means that the overall time spent in the iteration is measured exactly, not just as the sum of the time associated with the samples inside the iteration.

We apply a similar, but more fine-tuned approach, as illustrated by Figure 5.14: the sample interval in which a source code instrumented region begins is cut in two halves based on the time stamp of the start of the region. The first half stays outside the region (i_1), and only the second half will be attributed to the call path next hit by a sample inside the region ($i - 3$). Similarly, when leaving a region, the first half of the sample interval stays inside ($i - 1$), and the second half, together with the first half of the entering interval is attributed to the first call path hit outside the region ($i_1 + i_2$). This is a very small adjustment, typically at the scale of 0.01s per user region, but it can make a difference in case of very short user regions, and it guarantees that the time inside such a region or iteration stays exact.

Measuring OpenMP region times exactly

Measuring time in marked source-code regions exactly might sound like a minor issue at first, but it quickly becomes a question of major importance as soon as we start thinking about adding OpenMP support to the system. As an OpenMP region can essentially be handled as a source-code instrumented region, much like our iterations, the same solutions apply as well. And since these regions can often be very short, there is a good chance that statistical sampling will not be able to give a good estimate for the time spent in the region. If we also apply iteration instrumentation, this prevents the same OpenMP call path from improving its accuracy by collecting samples from a longer time period. In these cases, taking advantage of the exact timing information obtained from time stamps of the marked regions, while maintaining consistency with the sampling-based time information proves invaluable. As OpenMP support is beyond the focus of this work, we will not go into detailed analysis of this technique, but our preliminary results indicate that this technique provides the expected results.

5.4.2 Modified call-tree comparison methods

Another challenge for combining the hybrid sampling approach with the compression algorithm is that sampling inevitably provides non-deterministic results. Most of the metrics that the compression is based on are measured exactly, such as iteration time, time spent in different kinds of MPI communications and synchronizations, the counts of different MPI events, or *Bytes transferred*. One of the most important metrics however, *Visit count* is not measured exactly. In fact, for user call paths we only provide a lower bound, while for call paths leading to MPI communications we provide some limited guarantees. The only call paths where the *Visit counts* are exactly known are the terminal MPI call paths. Furthermore, sampling does not provide a guarantee of a complete call tree. Only the subtree with the call paths leading to MPI calls is guaranteed to be complete when employing our hybrid sampling scheme.

Under these circumstances, comparing call trees the same way as we do in compiler-instrumented measurements would be senseless. A number of modified call-tree comparison methods have been developed for this purpose. The common theme in these new comparison methods is that they compare only a subset of the call-tree nodes to avoid differentiating iterations based on call-tree features that are only statistically correct. The most straightforward

method is to compare the MPI visit counts only and ignore the call tree altogether. A more sophisticated approach is to ignore the purely computational parts of the call tree (the USR call paths, as introduced on page 22), but compare the structure and visit counts of the subtree consisting of the COM and MPI call paths only. This turned out to be overly strict in some cases, as the guarantees for the visit counts on COM call paths are not exact. The method we found most effective lies between these two extremes: compare the structure of the subtree consisting of the MPI and COM call paths, ignoring USR call paths, and compare visit counts on MPI call paths, but not on USR or COM call paths. This method leads to acceptable structural partition counts, typically less than six.

A side effect of the compression is that call trees of similar iterations are merged, potentially resulting in large numbers of 'phantom' USR call paths, coming from other similar iterations. This is an undesired effect when using direct instrumentation, but is actually useful in the sampling case. As we compared the MPI subtrees of the iterations before merging them, we know that they are as similar as possible, at least from the standpoint of their communication patterns. It is therefore likely that they are actually iterations with similar executed call trees. As sampling in many cases fails to collect the complete call tree of an iteration — especially when the iterations finish very quickly — getting a more complete call tree for the most often executed iterations is certainly a positive outcome.

After the decisions of the call-tree structure-based partitioning algorithm, the metric-based clustering algorithm takes care of the rest of the data compression. Our updated partitioning algorithm ensures that the partition count is not exceedingly high, which ensures that the clustering algorithm can easily adapt to the circumstances provided by the structural partitioning, using more clusters for partitions containing more varied iterations. In this way, using a high enough cluster count automatically results in a good representation of the original data, as we will see in the evaluation in the next section.

5.4.3 Evaluation

The quality of the results from this adapted form of the compression algorithm is first evaluated as an extension of the off-line prototype implementation, using the same kind of evaluation graphs as introduced in Chapter 4. No visual comparisons are included in this chapter, as Chapter 6, where we evaluate the actual on-line implementation shows such comparisons. It is more relevant to analyze those results in detail, as they are generated by the actual final implementation. The main point of the evaluation in this chapter is to provide error statistics undisturbed by run-to-run variation, using the off-line, post-mortem implementation.

As the distance between iterations is calculated the same way as in the direct instrumentation case, and only the structural partitioning algorithm has changed, we concentrate on this new aspect of the algorithm. To make comparisons between the evaluation of the direct instrumentation and the hybrid sampling method as easy as possible, we tried to keep the scale of the evaluation graphs the same as in the direct instrumentation case wherever possible.

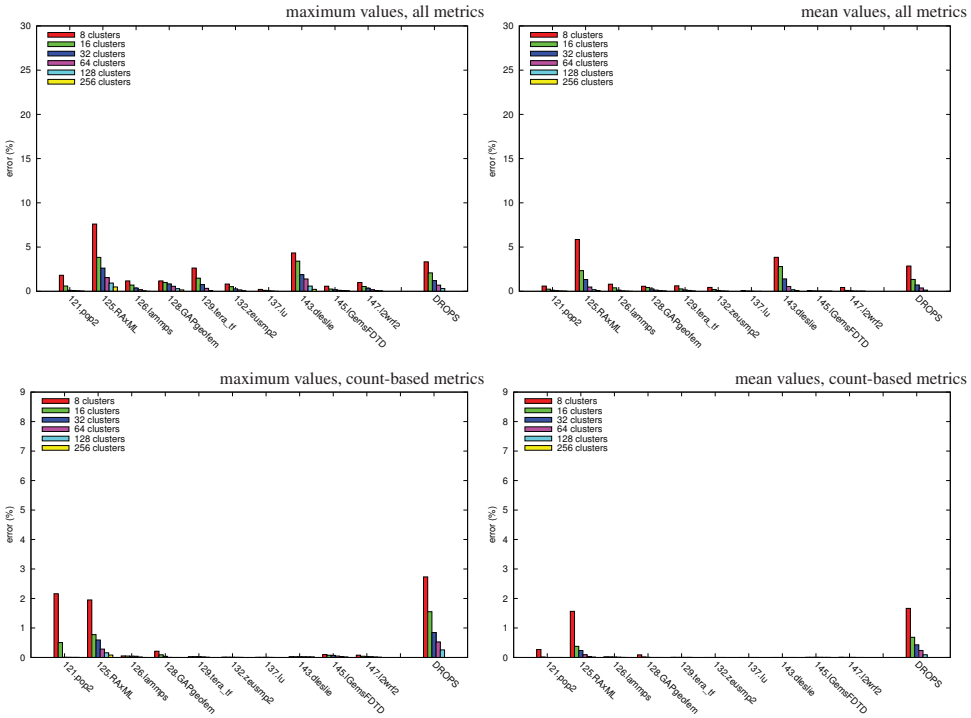


Figure 5.15: *Average error rates of the maximum and mean values when compressing hybrid sampling data.*

Error rates for entire iterations

Figure 5.15 shows the average error rates of the maximum and mean values in the graphs, both for all metrics and for the count-based metrics separately. Looking at the maximum error rates for all metrics, the overall trends are clearly as expected, using more clusters yields better accuracy. For most applications, we still get less than 1% error levels even at the lowest cluster counts, which is negligible for all practical purposes. There are no extremely high error rates in these graphs, the two worst cases are *125.RAxML* and *143.dleslie*, but they also get close to the 1% level at 64 clusters, which is a quite low cluster count given that these two cases both have several thousand iterations. *DROPS* is also one of the worse cases, but still at a quite acceptable error level. The error rates of the mean values are similar, just lower, much like we have seen them earlier in Figure 4.3 in the direct instrumentation case.

In the direct instrumentation case the error in the count-based metrics was zero for most applications, as those metrics contain no noise, are quite deterministic and correlate well with structural partitioning (see Figure 4.4 on page 58). In contrast, the same metrics have non-zero error for many applications when using the hybrid sampling method, which is a difference from the direct instrumentation case. This is because one of the most important count-based metrics, the *Visit count* is not deterministic any more under hybrid sampling, as the USR

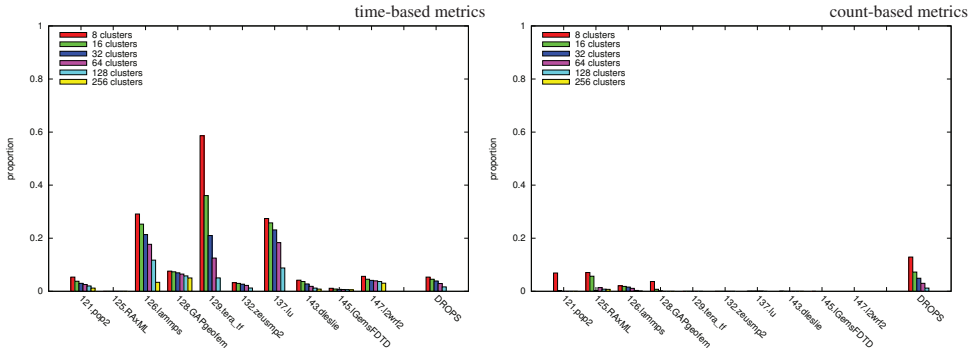


Figure 5.16: *Average error rates of individual call paths, when compressing hybrid sampling data.*

and in some cases the COM call paths are encountered a non-deterministic number of times, depending on when and from where the unwind operations happen. Still, these error values largely remain under the 1% level, and much lower than the errors for all metrics, indicating the same trends as in the direct instrumentation case.

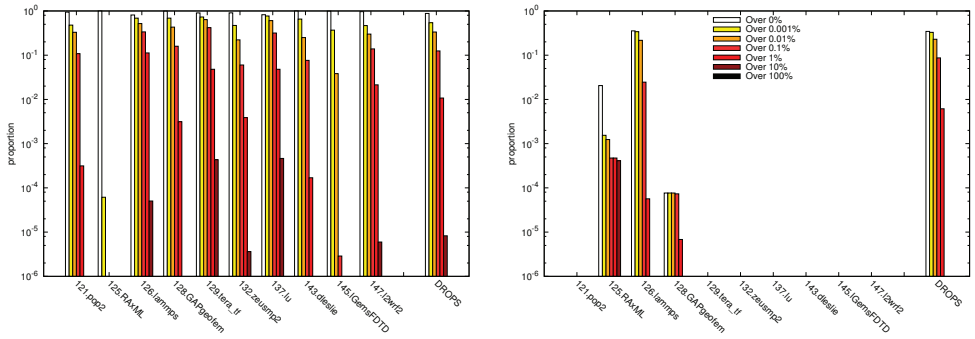
Error rates for individual call paths

Figure 5.16 shows average error rates of individual call paths. For time-based metrics, the trends are largely similar as in the direct instrumentation case before (see Figure 4.5 on page 59), *129.tera_tf* is still the worst case, but there are also differences, like the improvement in the *143.dleslie* case and the slight worsening of the *137.lu* case. Given that in this case we only evaluated error rates on COM and MPI call paths, as opposed to COM, MPI and USR in the direct instrumentation case, since we can not expect the same USR call paths to be present on call trees of equivalent iterations with hybrid sampling, these surprisingly low differences between the two cases indicate that the main characteristics of the compression remained largely the same when switching to the hybrid sampling method and the adapted compression techniques, which is certainly a positive result.

The error of count-based metrics in the direct instrumentation case was 0% for most applications (see Figure 4.6 on page 59). The results are similar with the hybrid sampling-based methods, as most of the error levels are very close to 0%. Exactly 0% error rates might not be realistic, as a few unwinding errors here and there are inherent to the new method, which introduces some slight variability to the *Visit count* metric on the COM call paths even where it would be completely constant between iterations.

Quantized call-path error rates

Figure 5.17(a) shows the quantized call-path error rates when using 64 clusters. Again, the trends are very similar to direct instrumentation (see Figure 4.7(a) on page 60), with the highest error rates occurring with slightly lower probability here. This is likely because we only take into account errors on COM and MPI call paths when evaluating the hybrid sampling method.



(a) MPI time-based metrics.

(b) MPI count-based metrics.

Figure 5.17: *Proportion of call paths in profiles reconstructed from 64 clusters having quantized error rates when compressing hybrid sampling data.*

Figure 5.17(b) shows the same quantized error rates in count-based metrics, where 4 applications show non-zero error. *126.lammps* is similar to the direct instrumentation case (see Figure 4.7(b) on page 60), with *125.RAxML* showing clearly larger errors, and *128.GAPgeofem* and *DROPS* being new additions. *125.RAxML* and *128.GAPgeofem* likely changed because of the slight variations caused by unwinding errors, while *DROPS* was expected to have relatively high variability by design, due to its use of automatic workload balancing. High, non-systematic variation in the count-based metrics makes them similar to time-based metrics, generally not predictable enough for perfect characterization.

Processing time

Figure 5.18 shows the absolute and relative average iteration processing times of the compression algorithm. We found in the direct instrumentation case (see Figure 4.8 on page 61) that most of the overhead was caused by call-tree comparisons, as opposed to distance calculations. As the algorithm adapted to hybrid sampling data only compares the COM and MPI call paths and ignores the USR call paths, which are the most numerous in most cases, processing times dropped significantly. Whereas *125.RAxML* had an overhead problem before, this problem is gone here. The only case with significant processing times is *143.dleslie*, which has many iterations, and many MPI and COM call paths. In this case, the overhead is around the same as before, which is still much lower than 1% at the 64 cluster case, rising to around 5% in the 256 cluster case.

Overview

Overall, it was expected to find some amount of difference between compressing directly instrumented and hybrid sampling data, but the differences were quite low, and in a number of cases in favor of the newly adapted algorithm. In light of these results, we can conclude that the adapted algorithm works equally well on hybrid sampling data as the original algorithm

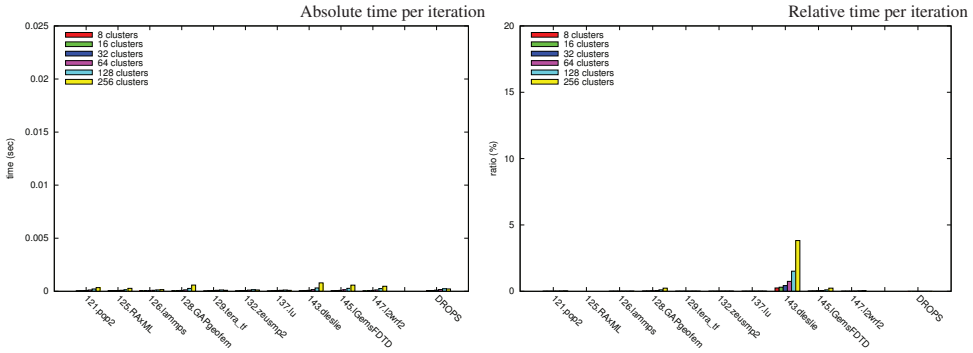


Figure 5.18: *Average time to perform the clustering of a single iteration.*

works on data from direct instrumentation, and can proceed to the evaluation of the actual on-line compression.

5.5 Summary

We have presented a novel hybrid approach for call-path profiling. This approach combines the low overhead of sampling with the detailed measurement of MPI routines possible with direct instrumentation. This enables the collection of all data necessary for the compression algorithm introduced in Chapter 4 to work properly, while avoiding potentially high overhead from small, frequently called functions. While the concept is straightforward, we found that multiple challenges arise in practice. First, we must unwind the stack even from the directly instrumented PMPI wrappers, since we can no longer observe (nearly) all calls to keep track of the changes of the current call path. However, the frequency of MPI calls requires several stack unwinding optimizations to keep measurement overhead sufficiently low. Second, we must ensure that the two measurement techniques do not interfere with each other. This challenge led us to develop a novel sampling methodology that accounts for the omission of samples that occur during MPI routines.

We implemented our hybrid profiling approach within Scalasca and presented a detailed evaluation of the costs to gather call-path profiles with direct instrumentation and our hybrid sampling approach. We found that both approaches work well for many applications in the SPEC MPI2007 suite. However, direct instrumentation suffers significant overhead for some applications, while pure sampling alone makes gathering some critical information — like the metrics based on MPI arguments — difficult. In contrast, our hybrid combination of direct instrumentation for MPI and sampling for the rest of the execution effectively overcomes these issues and provides a good combination of measurement quality, efficiency and ease of use. Our case study of the *DROPS* CFD application in Section 5.3 demonstrates that the detailed information and reduced overhead of our hybrid approach facilitates the analysis of real applications where it was impractical before. Overall, our approach significantly improves on existing techniques and we are working to make it available in a forthcoming release of the open-source Scalasca toolset.

In Chapter 6, we evaluate the on-line usage of the compression algorithm and discuss several examples including the *DROPS* code along with graphs of compressed measurements for visual evaluation. We give special attention to the comparison of the quality of data collected using direct instrumentation and hybrid sampling, and the compression quality achieved when applying the compression to data from the two measurement methods.

Chapter 6

Evaluation of On-line Compression

After the positive evaluation of the newly developed compression technique using the off-line prototype, the final step towards a complete solution was the actual on-line implementation of the algorithm in the framework of the Scalasca measurement system. The design remained largely the same between the two implementations, but certain aspects had to be adjusted to the general limitations of a highly portable production measurement library. An example is that the prototype was conveniently implemented in C++, while the measurement library is written in a limited subset of C to facilitate easier linking with C & Fortran applications. Also, we had to adapt to the existing characteristics of the data structures used in the measurement library, and implement a few new features not previously provided there. An example is the ability to remove call paths from the call tree when their iterations are eliminated by the compression, as previously call paths were only added, but never removed. Other than these kinds of implementation details, the algorithm and design remained largely the same as in the off-line prototype.

We found that prototyping such complex systems first in the highly flexible C++ environment provided by our post-processing tool infrastructure and doing the actual implementation in the much more limited measurement system code based on the experiences gained from the prototype worked out very well. Having such an exact design in mind and a good understanding of the implementation challenges in advance leads to robust, clear and efficient code.

6.1 Evaluation

We evaluate the on-line implementation of the compression algorithm using both compiler-based *direct instrumentation* and the newly developed *hybrid sampling approach* side-by-side. This has the additional benefit that we can compare the impact of the different instrumentation techniques and their associated overheads on the quality of the measurement data.

DROPS is not included in the set of direct instrumentation measurements due to its excessive dilation. The *PEPC* application is not included in the hybrid sampling measurements as the technique is currently not supported on the Blue Gene. Although a Blue Gene implementation of the stack unwinding infrastructure is under development, it has not reached the required level of maturity for complete measurements. For initial results, refer to [84].

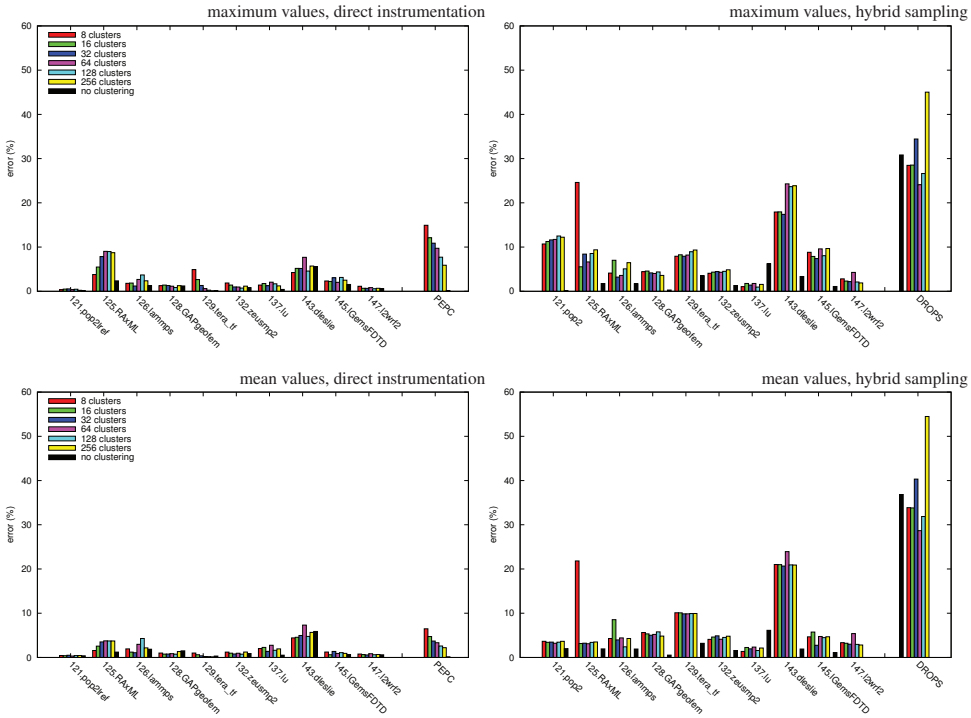


Figure 6.1: *Average error rates of the maximum and mean metric values for entire iterations, of all metrics, with direct instrumentation and the hybrid sampling approach.*

It is important to point out that both quantitative and qualitative evaluation is more complicated in the on-line case, as it is susceptible to run-to-run variation and noise. While in the case of the off-line prototype we could use exactly the same measurement to test every different compression configuration, in the on-line case every measurement is unique and they can not be expected to match exactly. Also in this case the runtime overhead of the compression algorithm itself can have an impact on the measured data, which it does not in the post-mortem evaluation. Based on our evaluation of the run-time and memory footprints of the algorithm in the off-line case we expect this impact to be relatively low, but it can not be assumed to be non-existent nor uniform on all processes being measured.

6.1.1 Error rates for entire iterations

A new element in our quantitative evaluation is the inclusion of a "no compression" result for every application. This shows the difference between two complete, separate measurements, which were not compressed. These measurements are the two best quality, lowest measurement dilation measurements from a population of five measurements, to minimize the chance of outliers. This is meant to characterize the impact of run-to-run variation on the given error metric for each application. Where this impact has the same magnitude as the error

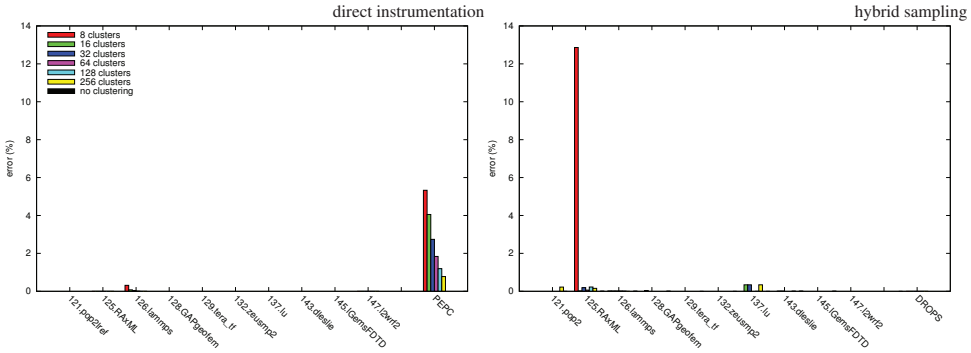


Figure 6.2: *Average error rates of the mean values of the count-based metrics for entire iterations, with direct instrumentation and the hybrid sampling approach.*

level indicated for the compressed data, that can be viewed as "perfect reproduction quality", as there is no way to do better than having broadly the same amount of difference from the reference data as that caused by simple run-to-run variation.

Looking at the error rates of the mean metric values for entire iterations, the black bars in Figure 6.1, the average error rates caused by run-to-run variation are around 1-2% for most test suite applications in the direct instrumentation case with the notable exceptions of *143.dleslie* which is over 5% and *PEPC* which is quite close to zero. A possible reason for *143.dleslie* having such high relative error is having a large number of very short iterations, which makes large relative errors out of small absolute differences. The same holds for *125.RAxML*, which shows a high error mainly in the error of the maximum values. The run-to-run variation of the *PEPC* measurements is expected to be lower in every way than that of the other applications, as the Blue Gene/P machine has inherently less run-to-run variation due to its specialized hardware and software stack (see page 1 for details).

Both run-to-run variation and the compression errors in the hybrid sampling case show higher error values than direct instrumentation. This can be expected of the inherently non-deterministic sampling approach, as unwinding costs from samples and even MPI events are expected to vary based on when and where in the code samples hit during the measurement.

In Figure 6.1, looking at the error rates of mean values in compressed measurements in the direct instrumentation case, we see that 8 out of 10 applications show error rates at the same level as the run-to-run variation. It is also true that there is apparently no clear trend towards less error with more clusters in most cases, except perhaps *129.tera_tf* and definitely *PEPC*. Overall, it seems that in most cases the run-to-run variation dominates the error rates of the compression itself. In the hybrid sampling case, the overall picture is somewhat worse, as most applications have around twice as much error when compression is used as their run-to-run error rates, with no clear indication of improvement with larger cluster counts.

Figure 6.2 shows that count-based metrics are still very well reconstructed in both direct instrumentation and hybrid sampling cases. Most applications show zero or negligible error rates in both cases, with the notable exception of *PEPC*, where we also did not expect perfect results due to its highly irregular patterns in count-based metrics, and the eight-cluster case of

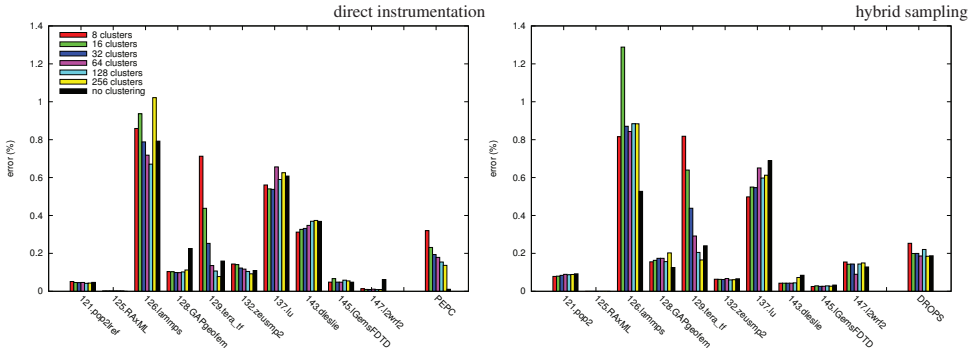


Figure 6.3: *Average error rate of time-based metrics for individual call paths, with direct instrumentation and the hybrid sampling approach.*

125.RAxML, which shows very high error rates in this regard. It is known that *125.RAxML* has a large set of fundamentally different iterations, which is a likely reason for eight clusters not being sufficient to represent these metrics well. The call-tree partitioning algorithm identifies nine partitions in the direct instrumentation case, versus only seven with the more limited comparison in the sampling case. It seems that those two partitions make a real difference in the count-based metrics for *125.RAxML*.

6.1.2 Error rates for individual call paths

Drilling down to the call-path level in Figure 6.3, we find generally higher error rates than in the off-line compression case again, which is not surprising. Still, most applications show less than 0.2% average error rate, which is on the same scale as the run-to-run variation. It is worth noting that only the communication subtrees consisting of COM and MPI call paths (as defined on page 22) are compared in the hybrid sampling case, as only that part of the call tree is deterministic and USR call paths are only statistically represented in the iteration call trees.

As expected, when we look at the count-based metrics only in Figure 6.4, we find substantially lower error rates for all applications except *PEPC*, which shows unusually complicated patterns in these metrics, which are harder to reconstruct exactly. It is also worth noting that in the hybrid sampling case we have more applications with non-zero error in these metrics, which is caused by the usage of less strict call-tree partitioning rules on the call trees.

6.1.3 Quantized call-path error rates

We compare quantized call-path error rates in Figure 6.5 in two different cases: in the first case, we compare compressed data with 64 clusters to a full measurement with no compression, whereas in the second case the comparison is between two full measurements without compression. This second comparison gives us a baseline measure for what amount of difference should be expected just based on the run-to-run variation. First of all, it is worth noting that these graphs show MPI time-based metrics only, which means that we are not

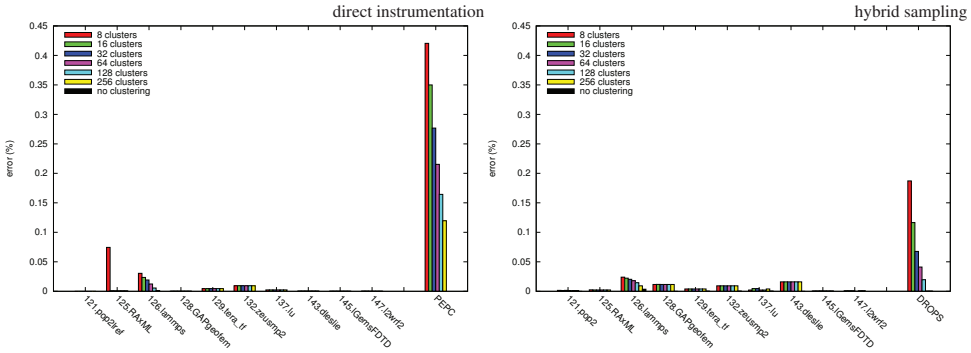


Figure 6.4: *Average error rate of count-based metrics for individual call paths, with direct instrumentation and the hybrid sampling approach.*

comparing timing differences on purely computational call paths here. It would not even be very meaningful in the sampling case, as there is no guarantee that the call trees match in two separate measurements. This means that we have a fair comparison between the two modes here, as this measure is meaningful in both cases.

The most important part of these graphs is the top region. It probably does not matter much if one call path in a million has a 100% error (e.g., if a value is erroneously doubled on some small, obscure call path). More important are the error rates that occur more often, on more than 1% or 10% of the call paths.

When comparing the two modes, it is very hard to declare a clear winner. There are differences, but they are mostly subtle, and each method wins over the other in some comparisons. Looking at the bottom (no compression) graphs, it seems that the pattern is mostly a characteristic of the application, and we get around the same kind of run-to-run variation irrespective of what method we use. The pattern depends a lot on how many different call paths there are, and what their scale is. Very small values can be expected to be more noisy in a time measurement. *125.RAxML* is a special case as it does not have point-to-point communication at all, so the MPI time differences can only show up on collective communication call paths, which are apparently much more deterministic than point-to-point calls. One possible reason for this is that when multiple collective communications are done, one following the other, the first one synchronizes the processes, and subsequent collectives will be able to start immediately since the processes are already synchronized, making the time spent in those calls much more deterministic.

The comparison between the two rows, using compression with 64 clusters and no compression is perhaps the most important here. As we have seen in earlier comparisons, 64 clusters tend to be enough for most applications to get a good representation of the original data in the graphs. Here we see that this is also true at the call-path level, as the differences after compression are at the same level as the differences caused by run-to-run variation. There is a clear exception from the rule, which was to be expected: 64 clusters are not enough for *PEPC*, where we see many more call paths having much more error than in the non-compressed case. As the previous recommendation from the off-line evaluation in Section 4.4 for *PEPC* was 256

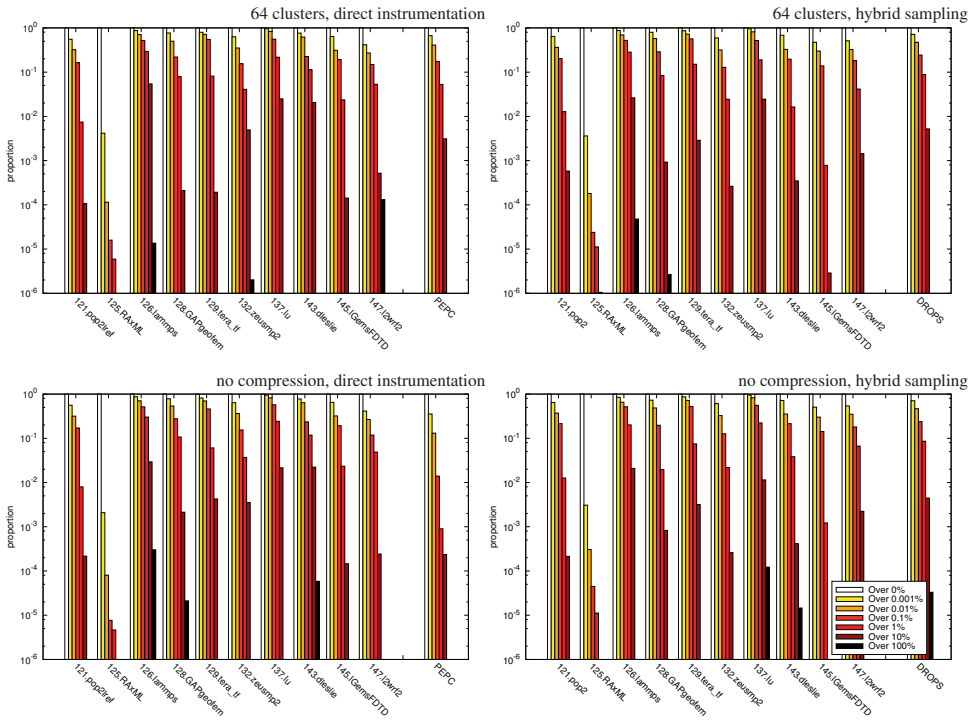


Figure 6.5: *Proportion of call paths in profiles having quantized error rates for MPI time-based metrics, with direct instrumentation and the hybrid sampling approach.*

clusters, seeing higher error rates when using 64 clusters here is understandable. This also shows that these comparisons have a potential for pointing out compression quality problems where they are significantly larger than the run-to-run variation.

Figure 6.6 shows the same comparison for MPI count-based metrics. As always, count-based metrics are much more deterministic and show no run-to-run variation, at least in the direct instrumentation case, as we see in the bottom left graph. In the sampling case, we see a very low amount of run-to-run variation in these metrics in the applications *121.pop2* and *DROPS*, which is certainly a bad sign, as no such variation was expected in these metrics. In the *121.pop2* case the differences are caused by measurement artifacts, while in the *DROPS* case they are inherent to the application.

The problem with *121.pop2* is not in our power to fix, as it originates in third-party code, during unwinding. Still, we are able to detect the occurrences of this problem and handle them in the most meaningful way possible. The problem is caused by two `MPI_Allreduce` calls relatively deep in the call tree, where the call-stack unwinding sometimes fails. Although this is very rare and occurs only on a few processes, it is enough for the MPI count-based metrics not to match up properly on these call paths. Of course we do not lose these MPI calls, and the overall values are correct if we add up everything, the difference is just that these few cases appear separately, in a separate section of the call tree for failed unwinds. It also does

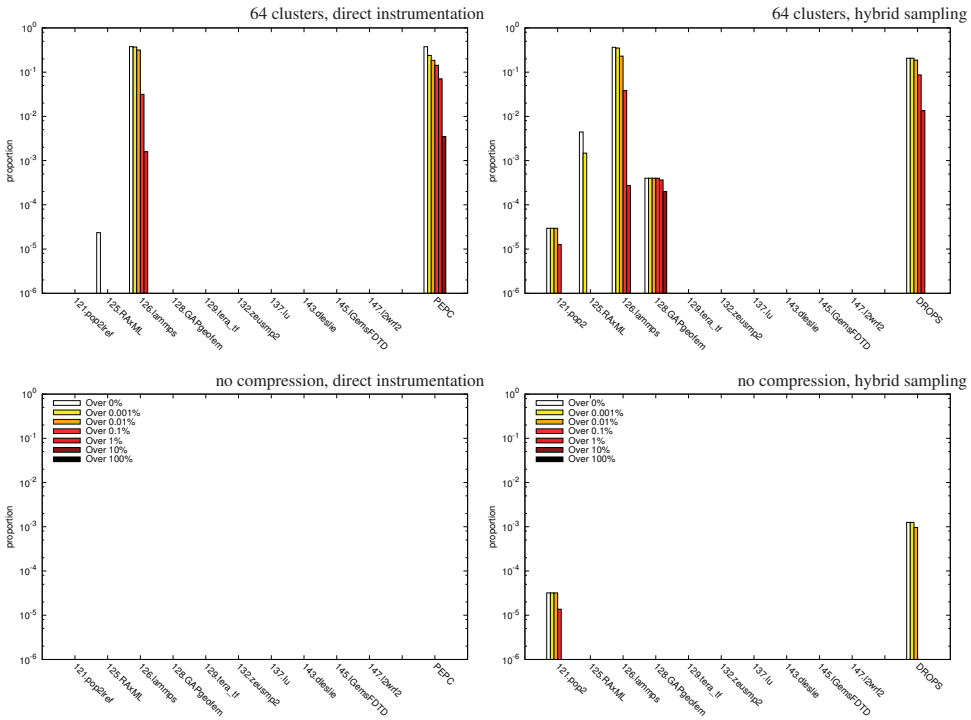


Figure 6.6: *Proportion of call paths in profiles having quantized error rates for MPI count-based metrics, with direct instrumentation and the hybrid sampling approach.*

not influence our analysis of the iterative behavior at all, as the problem only occurs in the initialization phase of *121.pop2*.

The *DROPS* case is perhaps even more interesting, as it shows a rare but existing phenomenon that we have not observed in any other application in our test suite. Here the differences are caused by the fact that the application's execution is non-deterministic, which is quite surprising at a first glance. We would more naturally expect multithreaded applications to be non-deterministic, but probably not MPI-based physics simulations without any multi-threading whatsoever. In fact, when we run the application with the same input data in the same batch job twice, it produces (very slightly) different results. As the developers confirmed, there is indeed non-determinism in the execution, whose source is inside the *ParMetis* parallel graph-partitioning library they use for their domain decomposition. The ParMetis library in turn likely gets the non-deterministic behavior from some not completely defined MPI operations, such as receiving messages in an undefined order from several source processes. It is a well-known fact that even just adding floating point numbers in a different order can yield different results, and this phenomenon is strong enough to lead to slightly different domain-decompositions, which in turn cause one iteration of the solver loop more (or less) to be executed based on these differences. We observe the first occurrences of such deviations after around 40 iterations of the main timestep loop, therefore this seems to be

the amount of computation required to aggregate enough uncertainty so that it shows such an effect. We could also verify our observations through the application's own monitoring system, as it reports a residual value after each iteration of the main time-stepping loop, which shows the same kind of slight divergence develop over time. The slight differences caused by this phenomenon clearly do not invalidate the results, but we should certainly reconsider our thinking about reproducibility and comparability of results in the light of these observations.

When 64-cluster compression is used (top row), we still get no errors from the majority of the applications, but *125.RAxML* and *126.lammps* show a small amount in the direct instrumentation case, joined by *128.GAPgeofem* and the *121.pop2* application, which has the aforementioned problem in the hybrid sampling case. *PEPC* of course is also an exception here, as it shows more significant error in the MPI count-based metrics when compression is used, since those metrics are extremely non-deterministic in the *PEPC* case.

A selection of characteristic applications from the test application suite are now considered in detail to establish and compare the quality of compressed data for the different measurement approaches.

6.1.4 Example: 128.GAPgeofem

128.GAPgeofem is a simple example with relatively constant behavior. Still, it is a good test case to see if the compression retains the abrupt change that occurs around iteration 500 properly. The answer is yes, even with eight clusters the main features of the communication behavior are already there, as seen in Figure 6.7 and Figure 6.8. It could be argued that using somewhat more clusters gives some more fine grained detail, but it also makes noise more visible. Of course, this is not necessarily a bad thing. It is also clear that there is some unwanted noise introduced when using the hybrid sampling approach, especially in the collective communication times. This is likely because of the additional operating system activity to maintain internal timers and deliver interrupt signals for sampling and the variable amount of time unwinding takes, which can introduce some workload imbalance that has to be synchronized at collective communication events. Still, looking at the graphs and maps in both modes, even the very low cluster counts give a good representation of the uncompressed data.

6.1.5 Example: 129.tera_tf

129.tera_tf is one of the more challenging examples with its gradually changing metric values over time, therefore we can not expect a very low cluster count to suffice here. We see gradually improving quality as the cluster count grows, but interestingly enough, we get quite good representations as early as 32 clusters in both Figure 6.9 and Figure 6.10, especially in the average values (blue) on the graphs. The maps also show the main patterns quite early on, although it is around 64 clusters when they become mostly indistinguishable from the uncompressed ones. Still, the main trends can be observed already with eight clusters, especially in the direct-instrumentation case. The hybrid sampling case needs more clusters

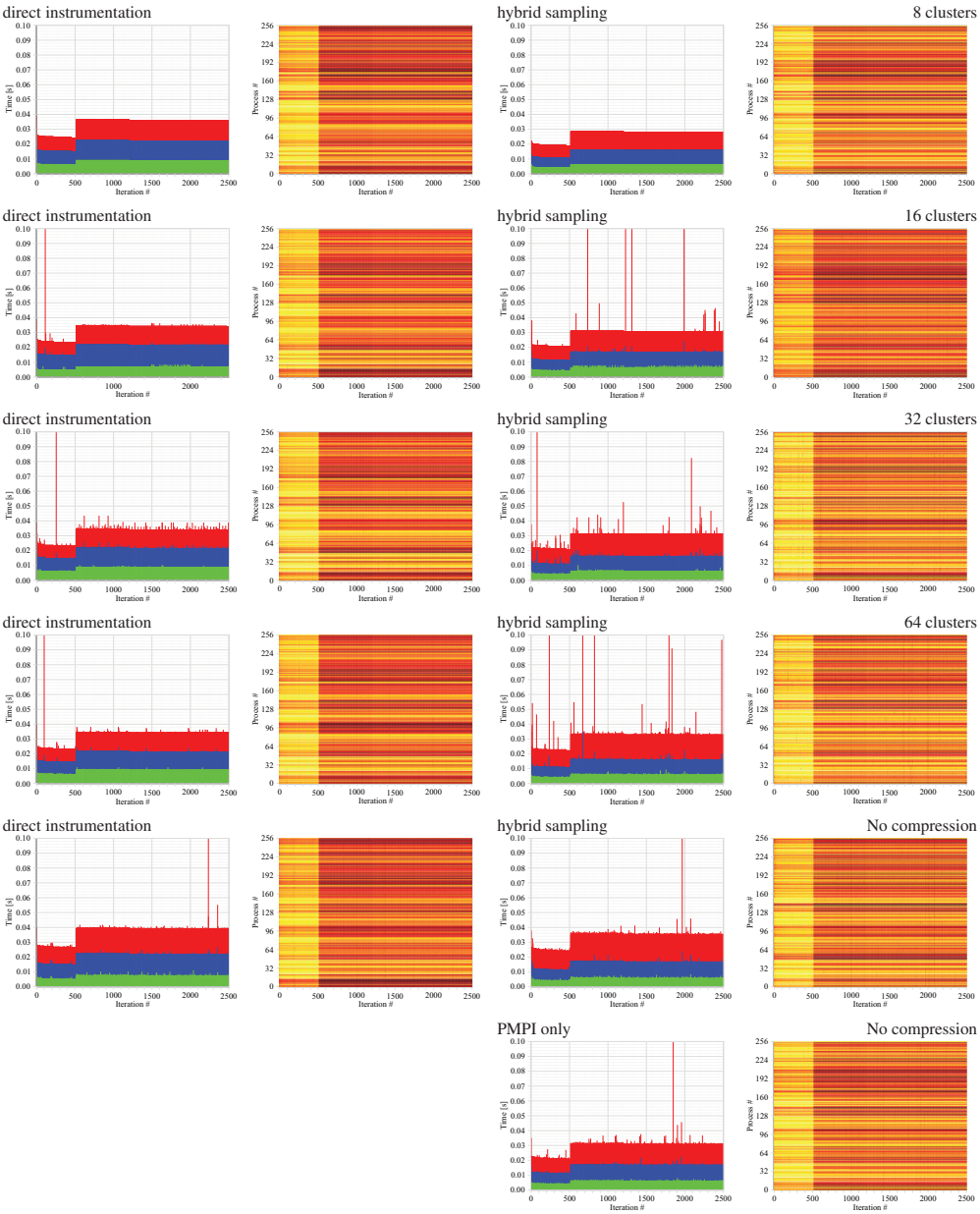


Figure 6.7: *Comparison of iteration graphs and value maps of MPI point-to-point communication time for 128.GAPgeofem with different cluster counts.*

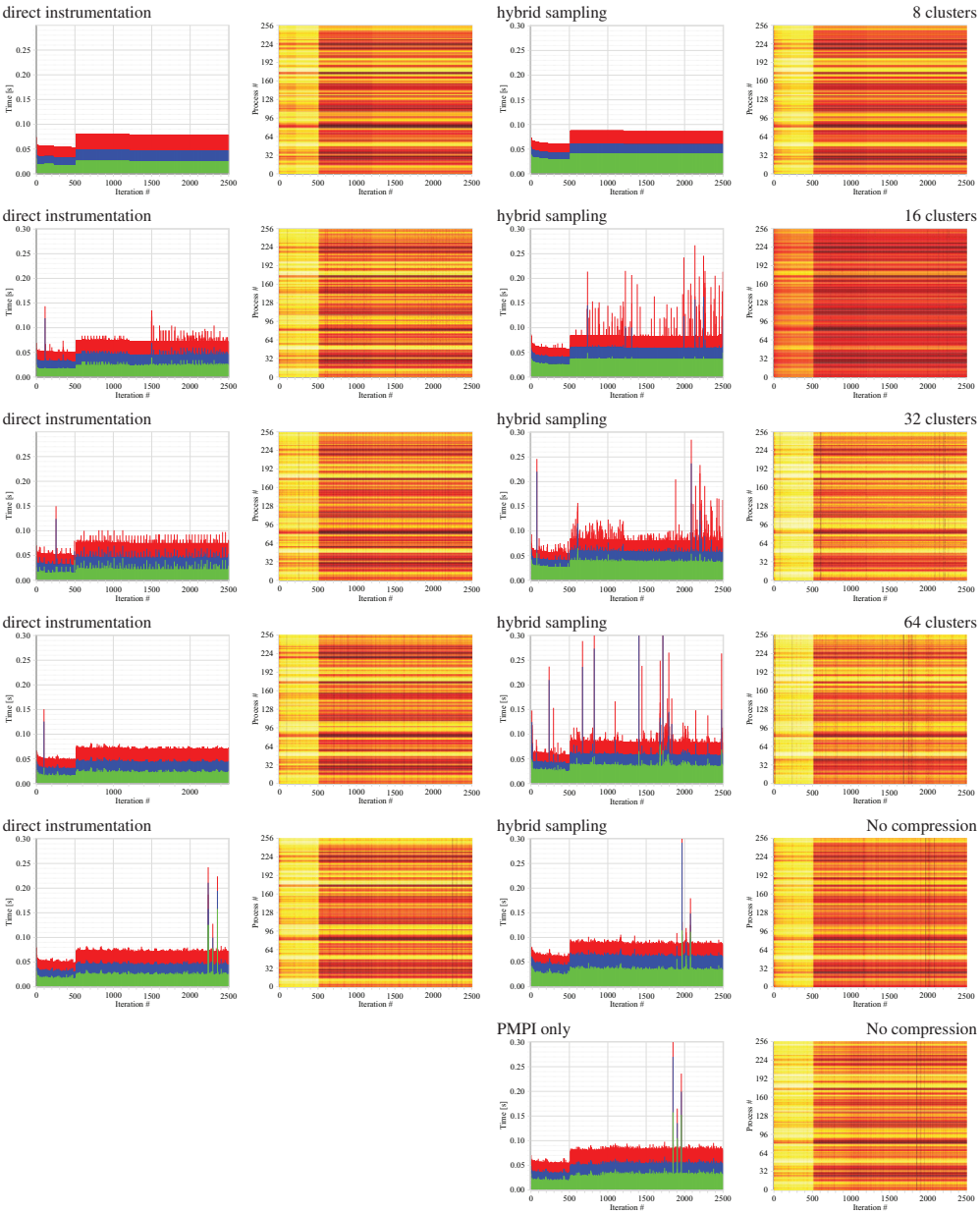


Figure 6.8: *Comparison of iteration graphs and value maps of MPI collective communication time for 128.GAPgeofem with different cluster counts.*

to represent the first few iterations correctly. Still, the overall picture is surprisingly well represented at relatively low cluster counts in both cases.

One more thing is immediately obvious when looking at these figures: The direct instrumentation results do not exactly match the hybrid sampling results. While the patterns match quite well, their magnitude is off by a factor of 2. We cannot observe the application execution without measuring it, and it is generally not possible to measure something without changing it, but we can reduce our impact by using as little instrumentation as possible. Using the PMPI wrappers only is the lowest impact solution in our case. As the figures clearly show, these measurements in the lower right graphs match the hybrid sampling measurements quite well. This is because the sampling approach has lower measurement dilation, as previously seen in Figures 5.4 and 5.5 on page 83, which means potentially less waiting time in communication calls due to imbalance caused by the instrumentation. Another important factor is that compiler-based direct instrumentation has the potential for inhibiting certain compiler-based optimizations such as inlining. This alters the behavior of the application, meaning that both measurements are perfectly valid, but the two executables are optimized differently. In this case the hybrid sampling approach is preferable, as its optimization level is closer to that of an uninstrumented executable.

6.1.6 Example: 132.zeusmp2

Figure 6.11 and 6.12 show the compression quality for *132.zeusmp2*, which is a similarly challenging application as *129.tera_tf*. It is somewhat more interesting, as the minimum values (green) are also prominent in Figure 6.11, adding more data to be represented on the graphs. With both techniques, relatively good representations start at the 16 cluster level, although the maps show the most important details already at the 8 cluster level. Note that there is some amount of noise present in the measurements, with significant noise peaks appearing somewhat more often in the hybrid sampling case.

The two measurement-collection methods differ in the magnitude of their results again, although the difference is not as large as in the *129.tera_tf* case. As the measurement using the PMPI-only version indicates, it is most likely that the hybrid sampling approach provides results closer to what data from an uninstrumented executable would look like if we could measure it without interference from the measurement. This is another example where the hybrid sampling method proves advantageous, as it gives the opportunity for measuring behavior more similar to an uninstrumented executable, while still providing much more insight than a simple PMPI wrappers only measurement.

6.1.7 Example: 143.dleslie

As Figures 6.13 and 6.14 show, and our earlier results in Figure 5.4 and Figure 5.5 on page 83 also suggested, *143.dleslie* is one of the rare cases where the hybrid sampling method causes significantly more overhead than direct instrumentation. *143.dleslie* is a hard example, not because of its behavior changing so much over time, but because of its large number of very short iterations. They cause all our metric values to be at a much lower scale than in other

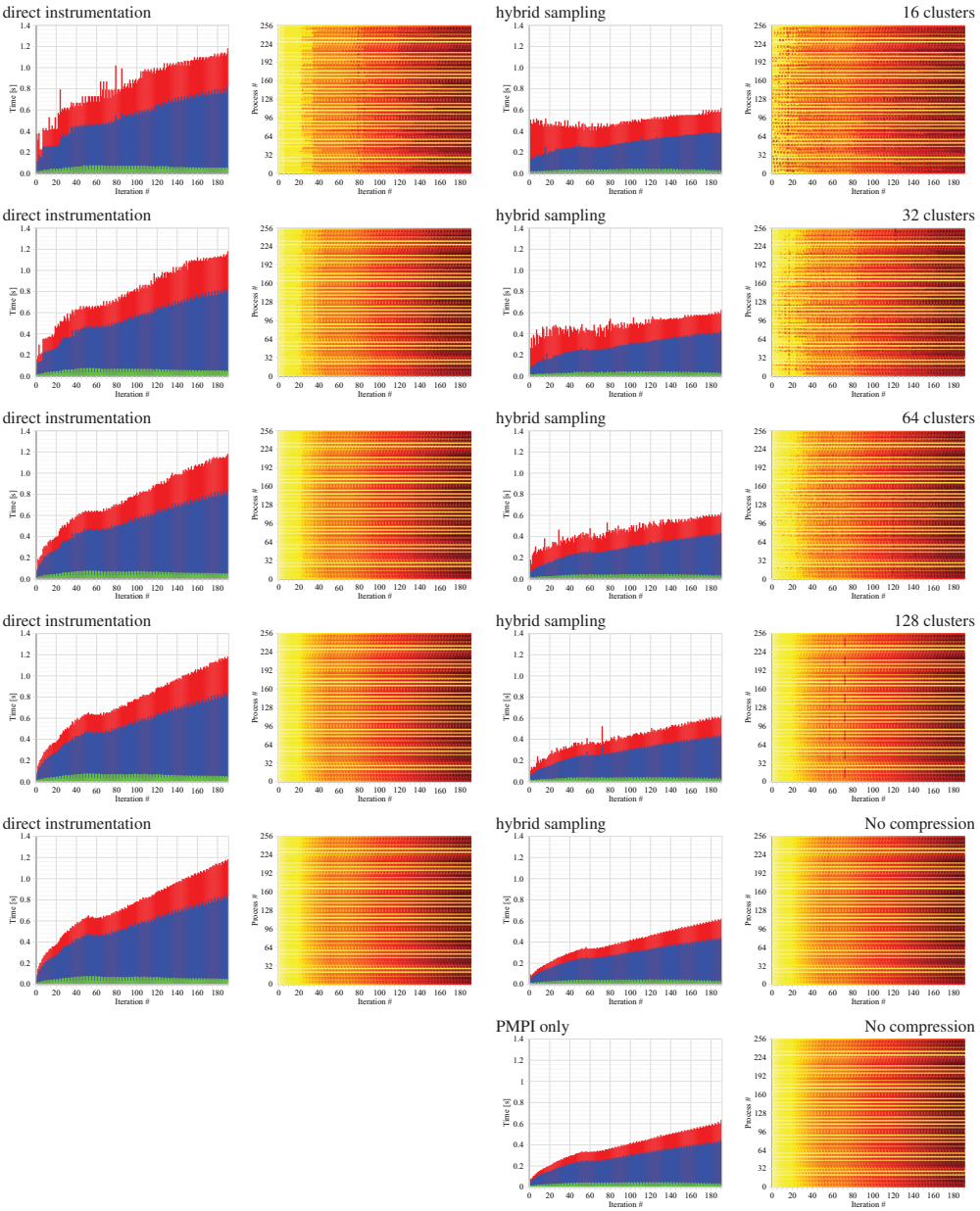


Figure 6.9: *Comparison of iteration graphs and value maps of MPI point-to-point communication time for 129.tera_tf with different cluster counts.*

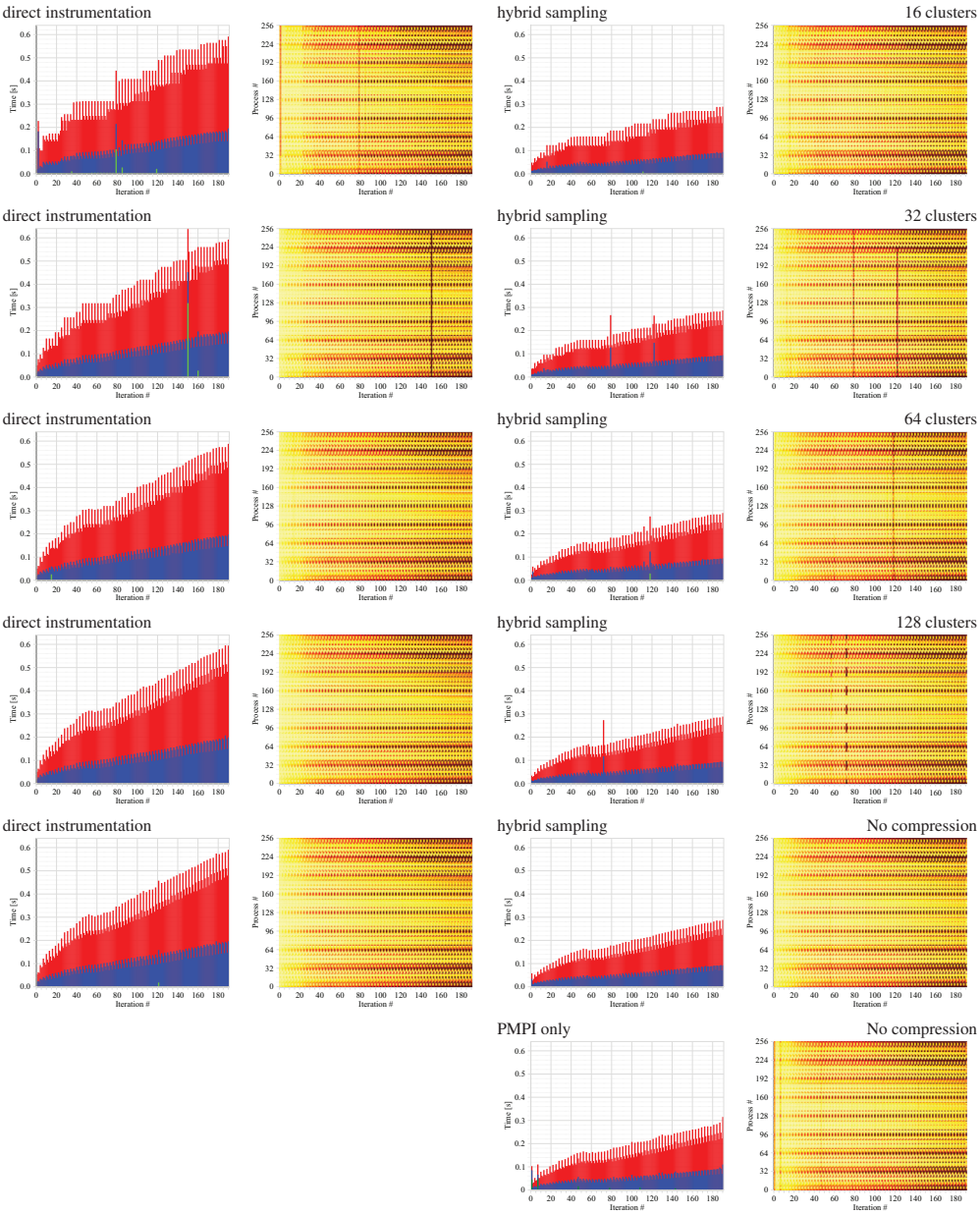


Figure 6.10: *Comparison of iteration graphs and value maps of MPI collective communication time for 129.tera_tf with different cluster counts.*

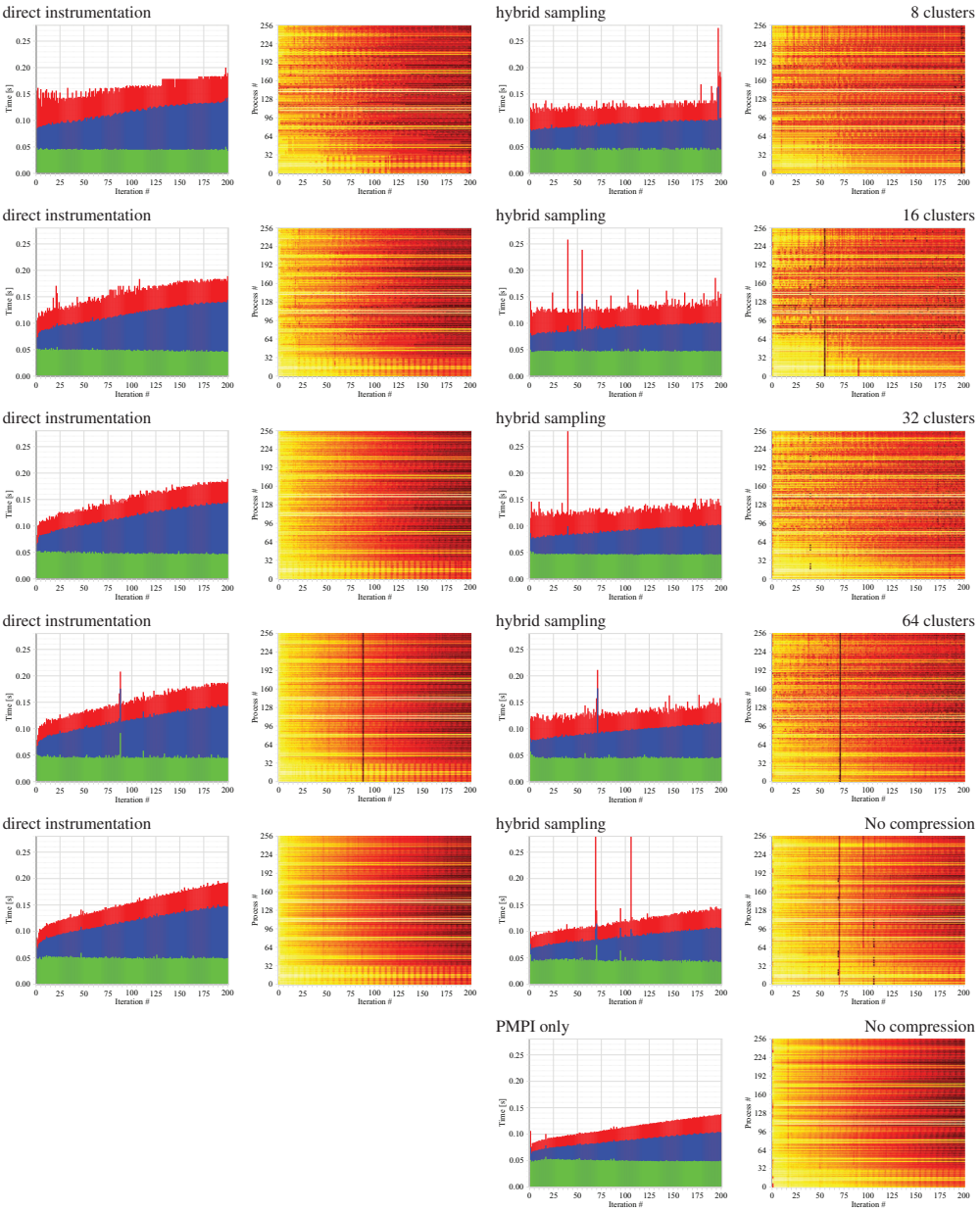


Figure 6.11: *Comparison of iteration graphs and value maps of MPI point-to-point communication time for 132.zeusmp2 with different cluster counts.*

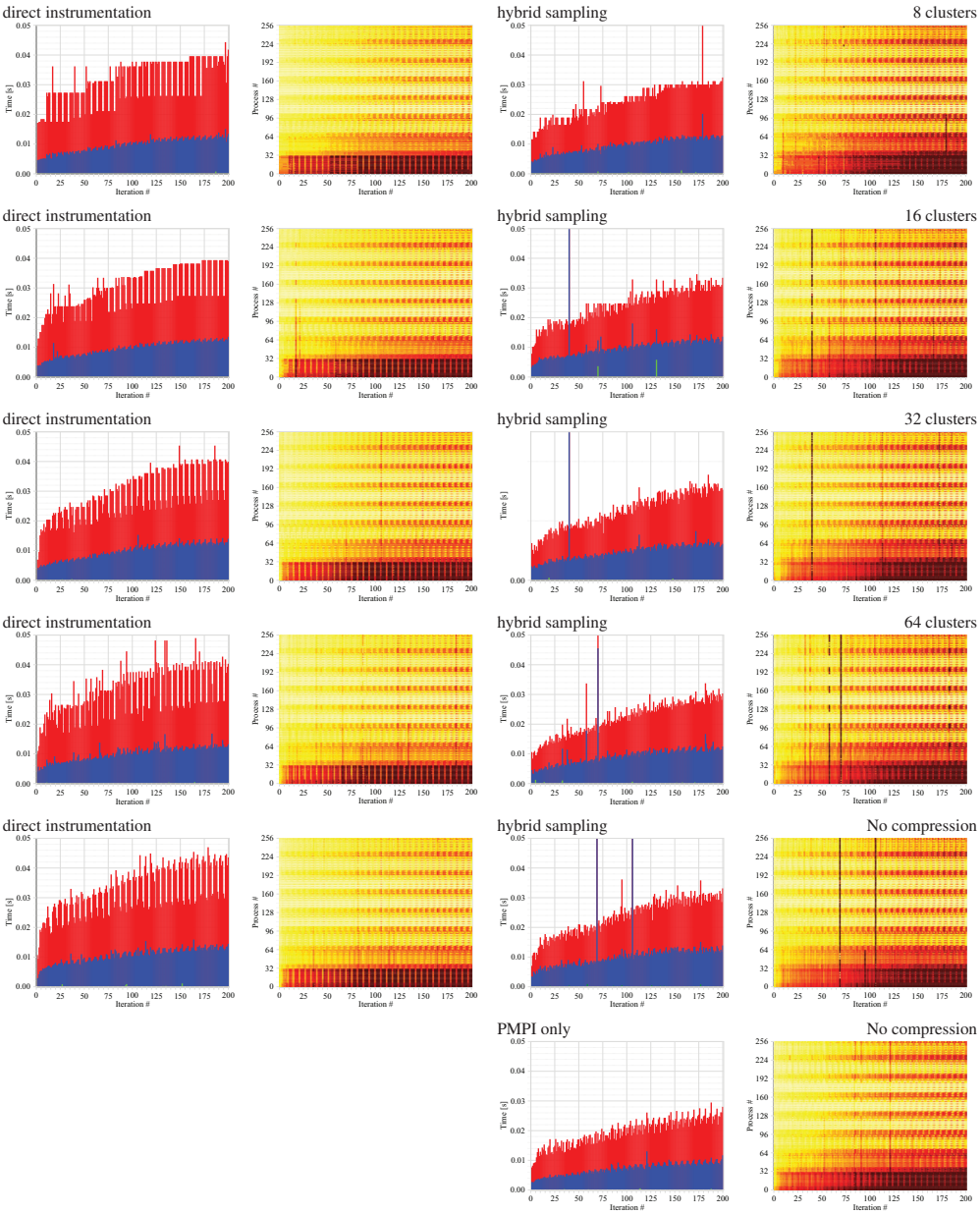


Figure 6.12: *Comparison of iteration graphs and value maps of MPI collective communication time for 132.zeusmp2 with different cluster counts.*

applications, making them more susceptible to noise. Combined with the hybrid sampling method's stronger tendency to introduce noise, *143.dleslie* becomes a good counterexample where using the hybrid sampling method is disfavored. Another argument against hybrid sampling in this case is that the measurements taken using direct instrumentation resemble much more closely the low-interference measurement using only PMPI wrappers. It is worth noting that in Figure 6.13 only the first 2500 iterations are shown, as otherwise the finer details would not be visible. In Figure 6.14 zooming in was not necessary, as not all of the iterations use collective communications and the graph is therefore not as dense as in the point-to-point communication case. It is also worth noting that in Figure 6.13, there are two main features on the maps: the 'horizontal' and the 'vertical' ones. Here horizontal lines correspond to differences among processes, while vertical lines correspond to differences among iterations. These vertical lines indicate that some iterations spent more time in point-to-point communication, and these iterations occur in a very systematic manner. The most puzzling feature of these patterns is that they are not shown in every measurement with the same magnitude. They are clearly visible in a number of measurements, but obscure or not present in others. In those measurements where they are very prominent, their much higher magnitude obscures the horizontal patterns in the variation among processes. It is not known what causes these variations between the measurements, but this illustrates the challenges of visualizing all the important information present in the measurements, as patterns with a larger magnitude tend to obscure others which might still be more important despite their lower magnitude. Variation among processes is likely to be more important here than iterations that take somewhat longer every few hundred iterations, as in the latter case only a select few iterations are affected, not the whole measurement experiment.

6.1.8 Example: PEPC

As previously mentioned, we evaluated *PEPC* on JUGENE, where unwinding was not available, so we can only show the direct instrumentation case here. Figure 6.15 shows the *MPI point-to-point communication time* and *communication count* metrics, to be comparable to our earlier results in Section 4.4. Testing the on-line compression of *PEPC* on the JUGENE system led to only partially satisfying results. Looking at the *MPI point-to-point communication count* graphs it is clear that the data provided by compressed measurements is in good agreement with the non-compressed data even at quite low cluster counts. The overall *Execution time* and other metrics also show good agreement. On the other hand, the *MPI point-to-point time* metric shows significant differences. The data seems to be converging to a certain pattern (which is very similar to what we have seen earlier in our tracing-based measurements), but the non-compressed profiling measurement does not match that pattern completely. Around half of the iterations are matching the pattern seen in all the other measurements, while the other half spends significantly less time in point-to-point communication. As we used the Blue Gene for *PEPC*, where system noise affects measurements to a much lesser degree, this result is also largely reproducible. Our most likely conclusion is that although the compressed data reproduces the temporal pattern with sufficiently high fidelity, the imbalance introduced by applying the algorithm perturbs the execution in a way that results in a somewhat different temporal pattern. This shows one of the main pitfalls of taking performance measurements in

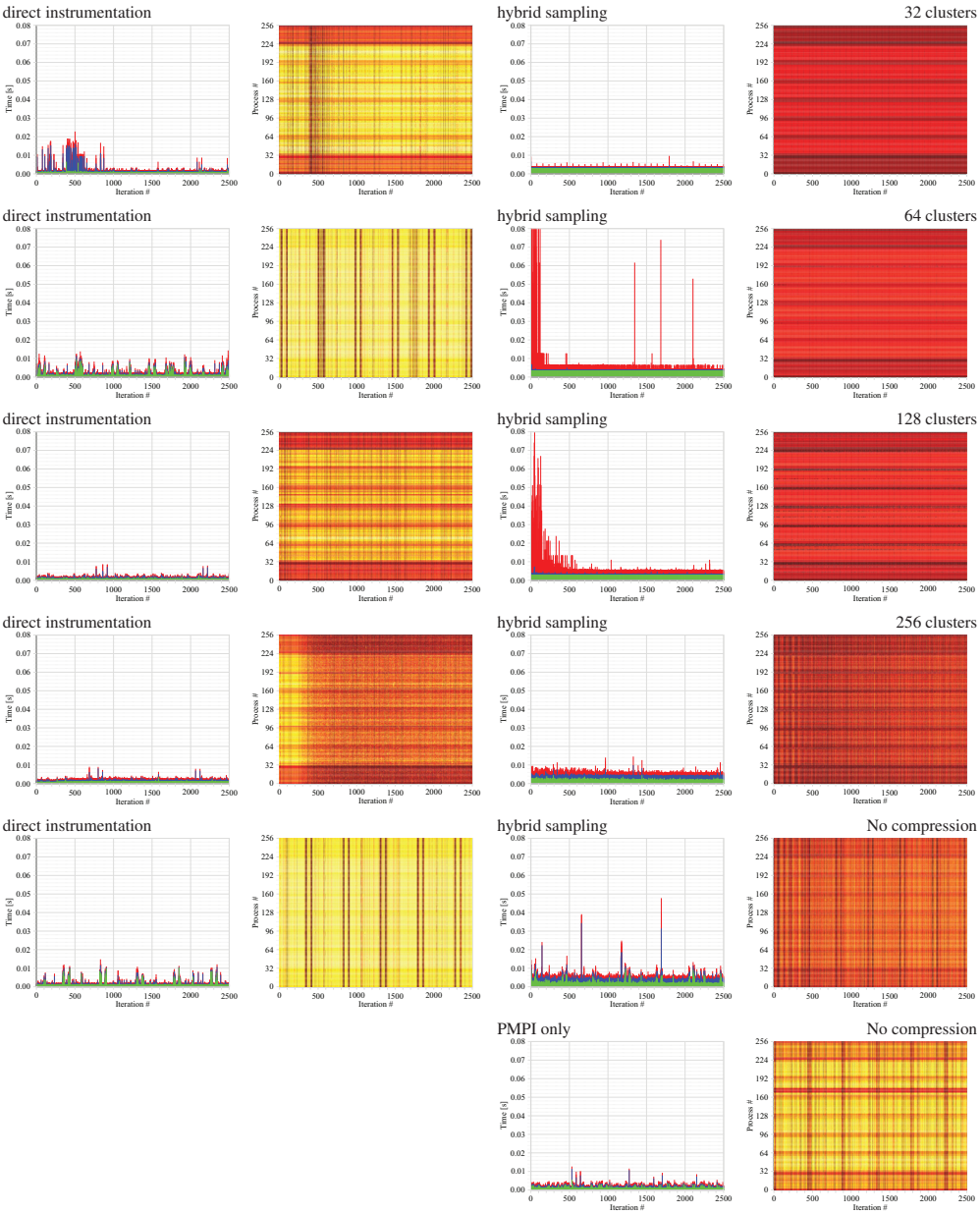


Figure 6.13: *Comparison of iteration graphs and value maps of MPI Point-to-point communication time for the first 2500 iterations of 143.dleslie with different cluster counts.*

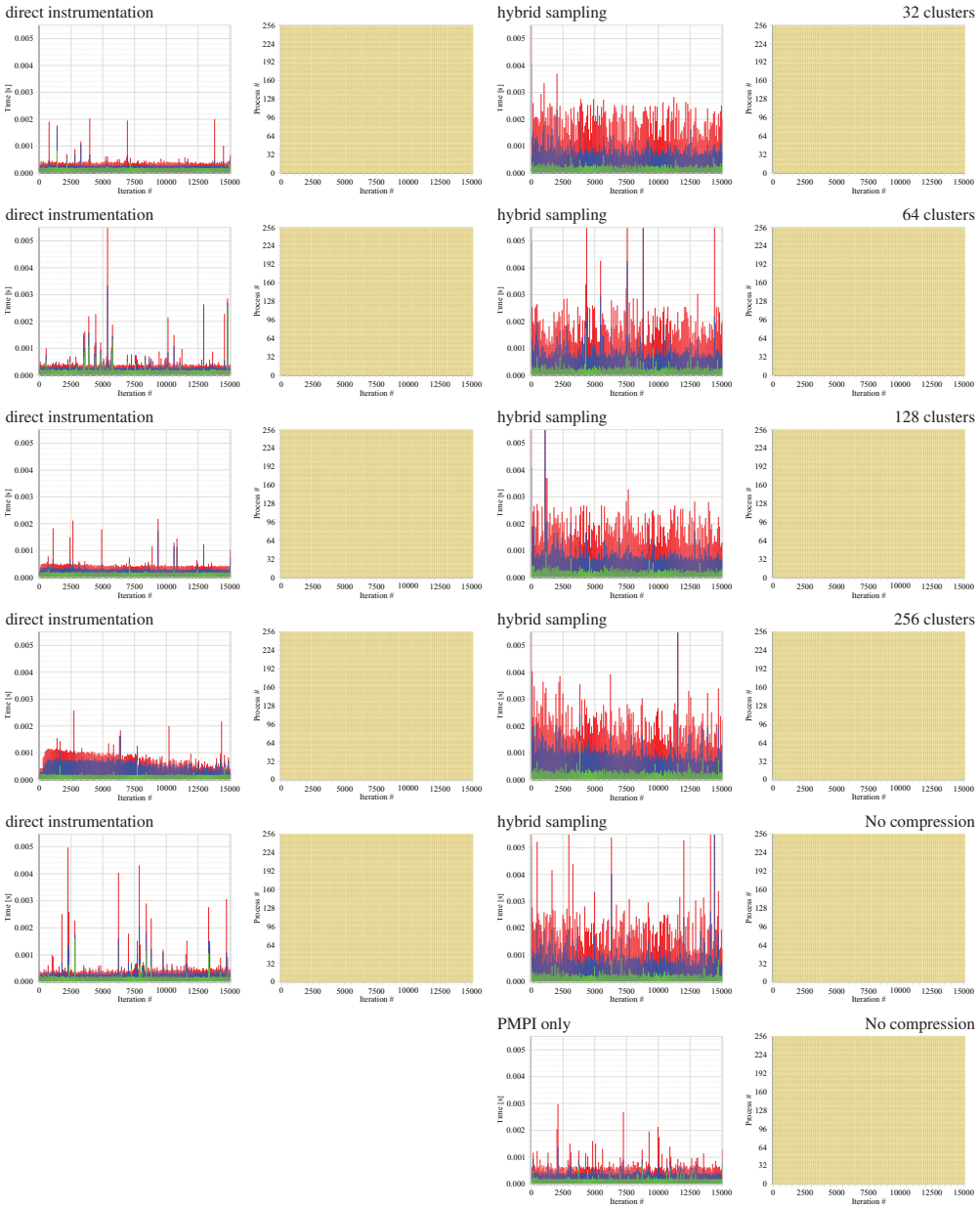


Figure 6.14: *Comparison of iteration graphs and value maps of MPI collective communication time for 143.dleslie with different cluster counts.*

general: by measuring a complex system, we also interact with the system in a very complex way, and there is no way to tell after taking the measurement how close our results are from the behavior we would see if we would not be taking a measurement. Great care must be taken both when setting up a measurement and when interpreting it. It is worth mentioning here that it is possible to reduce the impact of the compression algorithm on the execution by running it at global synchronization points, something we plan to implement for future versions of our solution.

6.1.9 Example: DROPS

Figure 6.16 shows the *MPI Point-to-point communication time* and *MPI Collective communication time* metrics at different cluster counts. In general, the reconstructed data shows good agreement with the non-compressed measurements, with all the important details, like the overhead of the reparametrization in every 20th iteration and the performance increase in the next few iterations clearly visible even when using lower cluster counts. However, when compared against the minimal-overhead measurement taken using the PMPI wrappers only, we see that the more heavy-weight measurement techniques did impact the measurement results, especially the *Collective communication time* metric, which generally tends to be sensitive to any sources of noise or imbalance because of their impact on waiting times in collective synchronizations. Still, this is a relatively easy case for compression, and the algorithm performed well even at lower cluster counts.

6.1.10 Overview

Figures 6.17 and 6.18 give an overview of data from compressed and non-compressed measurements. The clustering-based compression algorithm is applied at run-time. As the compressed and non-compressed cases are based on two separate measurements here, run-to-run variation is expected to introduce slight differences between measurements. Therefore, the best we can expect is that we get similar, but not exactly matching results. Note that the optimal cluster count — shown above each graph in bold — was used in every case, determined by empirical analysis. In general, we find that the results obtained with or without compression match quite well and the compressed data still represents the performance dynamics of the application reasonably well in every case. When using direct instrumentation, 64 clusters was enough for a good representation for all of the SPEC MPI benchmark codes. It is worth noting though that when using the hybrid sampling method there is some amount of additional noise introduced, which leads to higher optimal cluster counts, generally twice as much as in the direct instrumentation case. On the other hand, the lower measurement overhead of the hybrid sampling method often provides results closer to what is likely happening when no measurement is made, as in the case of *129.tera_tf*.

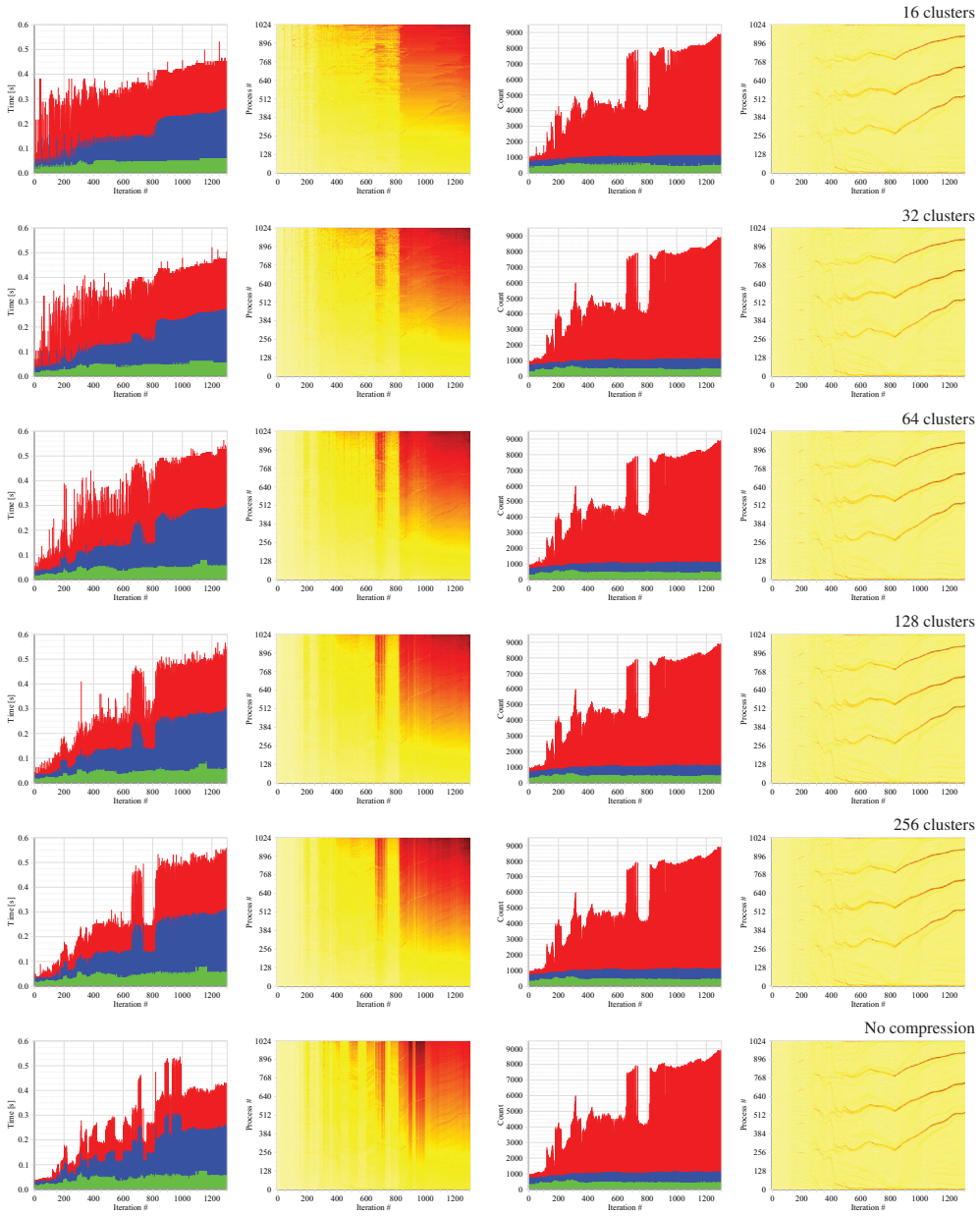


Figure 6.15: *Comparison of iteration graphs and value maps of MPI Point-to-point communication time (left) and MPI Point-to-point communication count (right) for PEPC with different cluster counts using direct instrumentation.*

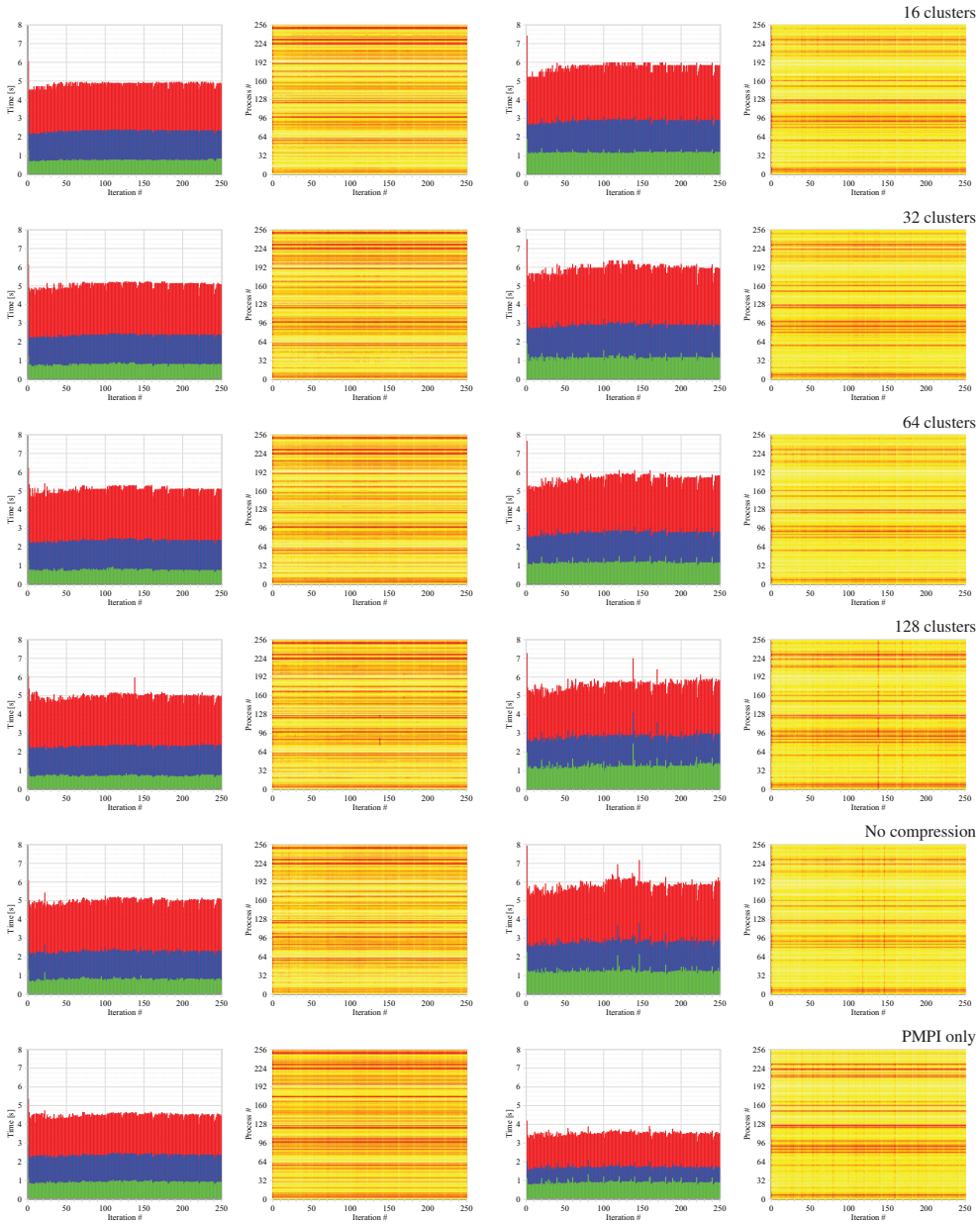


Figure 6.16: *Comparison of iteration graphs and value maps of MPI Point-to-point communication time (left) and Collective communication time (right) for DROPS with different cluster counts using hybrid sampling.*

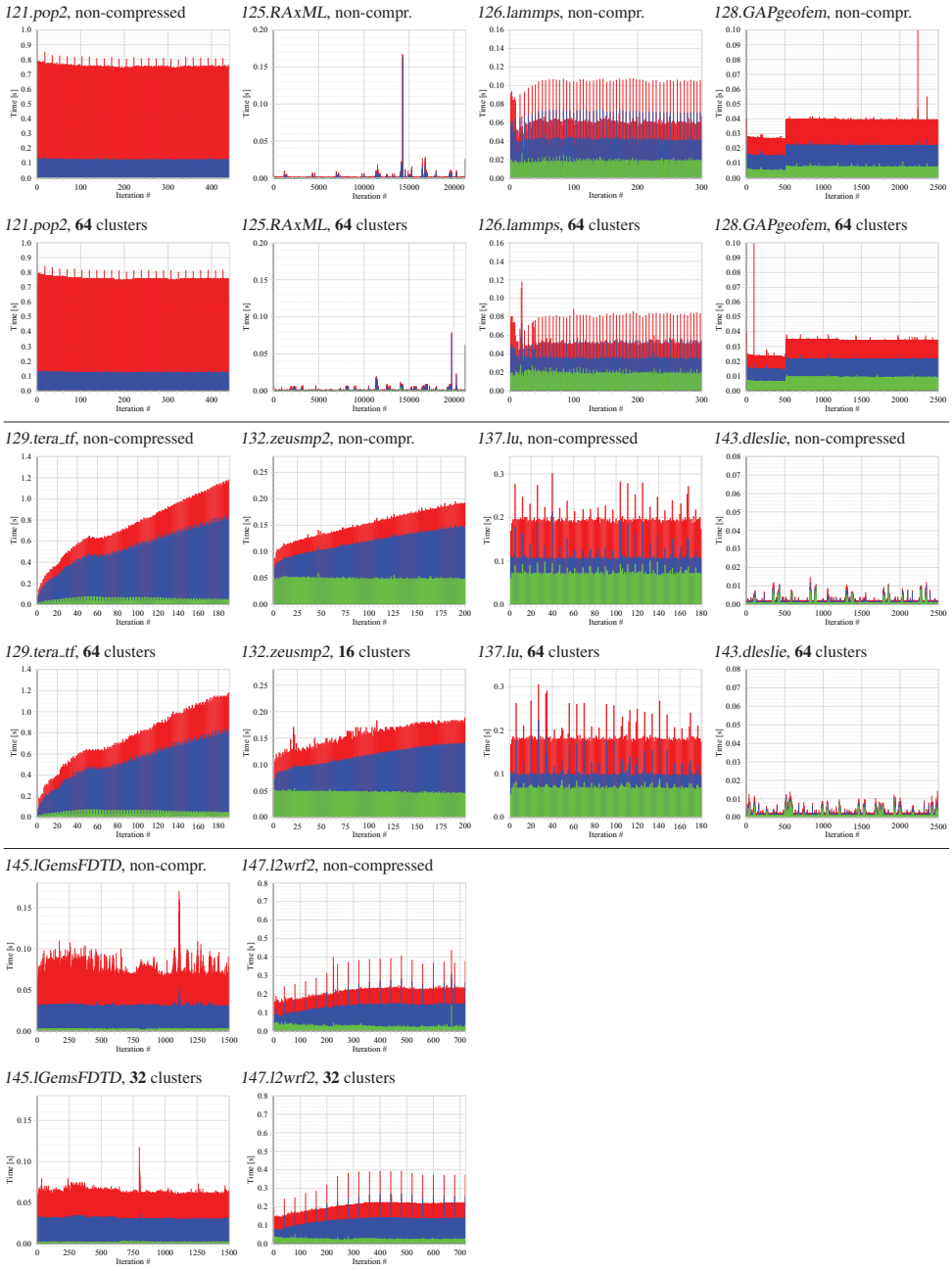


Figure 6.17: *Comparison of communication time iteration graphs without compression and reconstructed from acceptable compressed representations using direct instrumentation.*

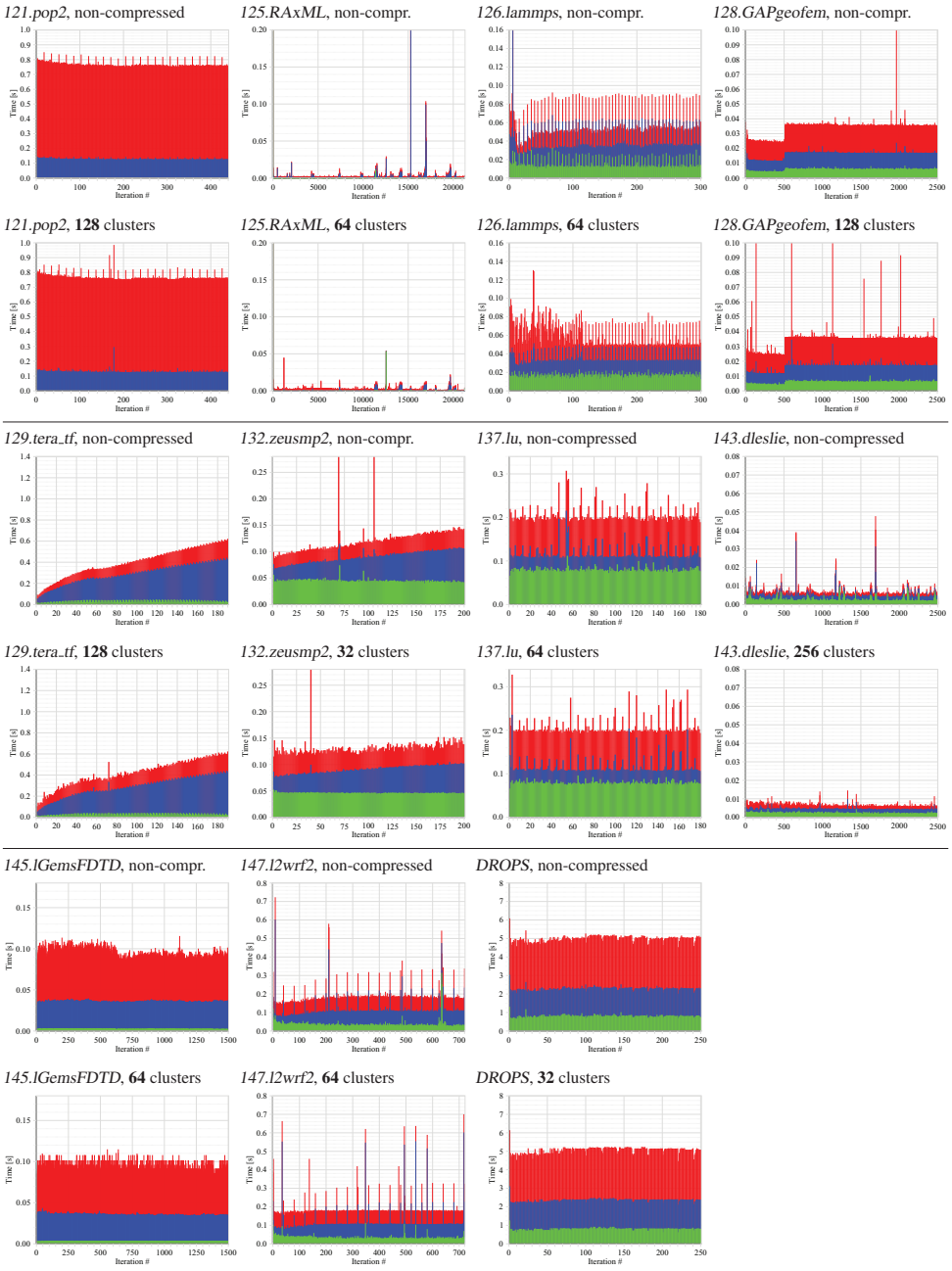


Figure 6.18: *Comparison of communication time iteration graphs without compression and reconstructed from acceptable compressed representations using the hybrid sampling method.*

6.2 Summary

In our evaluation of the on-line compression implementation we found that run-to-run variation and measurement noise is present in our measurements taken with both techniques, but to a greater degree when using the hybrid sampling method. Largely because of this variability, and the inherently less accurate call-tree comparison a generally larger number of clusters is required for a good compression when using the hybrid sampling method. Differences in the observed results when using the different measurement methods point out the importance of carefully assessing the impact of the measurement itself on the experimental results before jumping to conclusions. This consideration shows the value of having a more diverse set of measurement methods to choose from, as different applications need different treatment for the lowest overhead and best measurement quality.

In the next chapter we introduce related work. While some of the work related to the basic measurement techniques has been covered in Chapter 1, we discuss work related to our more advanced concepts here, in this separate chapter.

Chapter 7

Related Work

Work related to the basics of this work, including MPI, OpenMP, call-path profiling and tracing, as well as the main serial and parallel performance tools is introduced in Chapter 1. In this chapter we discuss what has been done in the field related to each of the main topics introduced in the previous chapters.

7.1 Performance dynamics

Perhaps the most straightforward approach to analyzing the evolution of performance behavior over time is by collecting trace data and visualizing the temporal changes of certain metrics like hardware counter values as in Vampir [68]. Traces can also be used to display timeline views differentiating between different communication and computation regions using tools such as Vampir or Paraver [54]. Regions of interest, such as main execution phases or iteration loops can be determined automatically or marked manually through iteration or phase instrumentation. The term "phase" has been defined and used in numerous papers in a number of different ways. An early occurrence of the term's usage in parallel performance analysis can be found in the 1990 paper describing the prototype of the *Paradyn* tool, where the authors talk about "periods of time where some combination of performance metrics maintain consistent values" [61]. In a subsequent paper they show an example of marking execution phases manually [42]. The Oracle (formerly Sun) Studio Performance tools provide an API for delimiting intervals in the program execution, providing coarse-grained overview statistics of each interval separately [92]. Phase profiling in the *TAU* performance system allows the user to obtain a separate profile for every marked program interval, distinguishing between static and dynamic phases [58]. Whereas performance metrics are aggregated across all executions of a static phase, a separate profile object is created for every instance of a dynamic phase. According to this distinction, marking the main time-stepping loop to measure the iterations separately, as we do in this work, can be classified as collecting instances of a dynamic phase. Similar in spirit, incremental profiling in the OpenMP profiler *ompP* takes a separate profile for every fixed-sized execution time interval, using elapsed wall-clock time instead of the dynamic program structure as the delimiter between phases [21].

7.2 Compression of time-series profiles

The principle of dividing the execution of a program into intervals and grouping them into phases according to their performance characteristics has already been studied by Sherwood et al. in the context of microprocessor hardware simulation [81]. Using clustering algorithms, they identified representative subsections of the instruction stream of a program that can be used as input for simulations that would otherwise be too slow if fed with the complete stream. The results of these shorter simulations can subsequently be extrapolated to reflect the execution of the entire program. In the context of large-scale parallel applications, various statistical and data mining techniques have been employed to reduce the size and improve the understanding of performance data. While projection pursuit [89] and clustering [69] have been applied to improve the understanding of real-time performance data, Ahn et al. have shown that multivariate statistical techniques provide a useful means to identify correlations between hardware performance metrics and to highlight clusters of similar behavior in process topologies [4], which could be used for compression in a similar way as we do. Moreover, PerfExplorer [43] structures profile data of parallel programs post-mortem by performing hierarchical and k-means clustering on vectors of metric values whose elements correspond to program regions.

Space limitations of highly-structured performance data sets that include a time dimension are most apparent for event traces containing timestamped records for a huge number of program actions. Complete call graphs are able to compress event traces by exploiting repetitive patterns [52], but require their prior existence at full length, as runtime overhead prevents the method from being applied on-line. Similarly, automatic structure extraction starts from very large trace files, explores their internal structure using signal processing techniques, and selects meaningful parts [12]. Whereas the previous approaches target the time dimension, Gamblin et al. lower the overall trace volume by tracing only a subset of the processes [23], with sample size periodically readjusted based on summary performance data sent to a central client process at runtime. ScalaTrace uses a combination of intra-node and inter-node compression techniques to provide a concise representation of large-scale traces [70].

Finally, application signatures provide a way of summarizing the time-dependent behavior expressed in historical trace data in a much more compact representation [57]. Here, the temporal evolution of a metric vector is described as a metric trajectory using curve fitting as compression mechanism, simplifying the comparison between two signatures.

7.3 Combining sampling and direct instrumentation

Individually, both statistical sampling and direct instrumentation are used in a wide array of tools. HPCToolkit [3] is one example of a tool that exclusively relies on sampling to deliver a comprehensive performance profile to the end user. It implements call-path profiling by gathering stack traces at each sample. It uses this information to map the profile data back to the user's source code and provide dynamic execution path information. In order to keep the overhead low, HPCToolkit applies a trampoline-based prefix optimization [20, 87], similar

to the thunk optimization presented in Section 5.1.1. Arnold and Sweeney implemented a similar technique earlier in a sampled call-path profiler for Java Virtual Machines [6], but their approach required virtual machine support for marking return addresses with special bits.

Direct instrumentation can be used for both profiling (i.e., providing aggregate information over the runtime of the process) and tracing (i.e., delivering all events as they occur during the execution of the program). Closely related to our work is *mpiP* [88], a profiler for MPI operations. Like our system, it uses the PMPI profiling layer to gather basic statistics about the application's MPI usage, and call-stack unwinding to provide call-path context.

On the tracing side, tools such as *strace* [55] provide per-process trace information. For parallel applications, tools such as Vampir [68], Jumpshot [13] or the Intel Trace Analyzer and Collector [45] record all message transfers between MPI processes and store them to disk complete with timestamps. They can then display the collected events on a timeline. This approach allows the user to carefully examine each individual message event, but it comes at the price of large storage requirements for the detailed trace files.

So far, little work has combined these two paradigms. However, several tools exist that offer both sampling and direct instrumentation side by side. The standard UNIX tool *gprof* [35] uses direct instrumentation to build an estimate of the call graph for sequential programs in addition to sampling for a separate flat profile. In the area of parallel performance analysis, multiple tool sets offer both approaches in a single environment. CrayPAT [15] has mutually exclusive sampling and direct instrumentation modes. Open|SpeedShop [77] and the Oracle (formerly Sun) Studio Performance Tools [47] simultaneously record sampled and instrumented events, but ultimately present separate analyses. Only recently and concurrently to our work, TAU/ParaProf [8] started to prototype integrating sampling and instrumentation annotation events in a single measurement and to visualize them together [62]. However, the TAU approach records profiles and traces separately from the two sources of information, and is only able to merge them after measurement. By comparison, our work integrates the two sets of events into call-path profiles as they occur and carefully accounts for their interaction. Another interesting combination of the two techniques can be found in Extrae [30], which adds samples to traces based on direct instrumentation to provide a more detailed view of the evolution of hardware counter metrics inside the instrumented regions. It applies a folding mechanism to derive fine-grained data from coarse-grained sampling by superimposing trace data from similar phases based on iteration instrumentation [32] or clustering of computation phases [31].

Chapter 8

Summary and Outlook

Over the course of this work, we touched a variety of topics and gained new insights into the challenges of performance analysis on present day HPC systems. Chapter 1 introduced the field and showed why scalable parallel performance analysis tools are essential for supporting the tuning and scaling of HPC applications. In the end of the chapter, the analysis of the *Sweep3D* application served as an example showing the importance of the analysis of time-dependent behavior in understanding complex performance phenomena in HPC applications.

In Chapter 2 we introduced the set of HPC applications used for evaluation purposes throughout this work. We used Scalasca to analyze the execution performance of these applications on two HPC platforms, and determined that the current release of Scalasca and its direct instrumentation approach was efficient and apparently accurate for most but not all codes. Applications with frequent calls to small routines suffered unacceptable measurement dilation from direct instrumentation.

In Chapter 3 we applied iteration instrumentation to these applications, taking advantage of the fact that most (but not all) of these HPC applications have a clearly distinguishable main timestep/iteration loop. Instrumenting these for separate measurement/analysis revealed a variety of interesting dynamic behaviors in configurable iteration charts and value maps of different metrics. This opened up an important new analysis dimension. The new measurement dimension also introduced a new problem: measurement storage/post-processing/presentation costs being proportional to the number of iterations. However, there are often considerable similarities between iteration executions.

In Chapter 4, applying our novel clustering-based compression algorithm to time-series call-path profiles demonstrated to offer significant reductions in measurement storage/post-processing costs. Our compression algorithm is necessarily lossy, however, with suitable methods effective reconstruction of complete iteration charts and maps is possible (even for challenging cases such as the *PEPC* application). Typically 64 or fewer clusters are required for acceptable fidelity, and when necessary more can be used to reduce the number of measurement artifacts.

Chapter 5 introduced a newly developed alternative measurement technique that we investigated to overcome the problem of excessive measurement dilation from direct instrumentation in certain applications. Moreover, we needed a solution that despite its lower overhead still provides full information about communication events, as these are important for the analysis of the results, and also essential for the compression algorithm to work. The solution

is customized to appropriately combine direct instrumentation of MPI (using standard MPI wrappers) and sampling driven by POSIX interval timers. Call-path profiling required careful unwinding of call-stacks from both samples and PMPI wrappers, and sophisticated integration of both types of measurement. The new hybrid, sampling-based measurement method has its limitations: the measurements are more challenging to interpret since they are partially only statistically accurate and subject to increased system noise; there is a significant number of call-stack unwind failures; in some cases even the highly optimized unwind operation is too expensive to be used from every MPI event; and the additional system dependencies restrict portability/availability. Still, this is a valuable complement to the purely direct instrumentation approach and essential for the analysis of certain applications. It is easier to use than direct instrumentation, as there is no need to set up function filters, and provides generally low measurement dilation, even in the cases where full direct instrumentation was problematic.

Our compression algorithm can be applied independently on each process during measurement, adding each iteration to the compressed dataset immediately after they are generated, and realizes considerable savings in measurement storage. In Chapter 6, comparing measurements based entirely on direct instrumentation to those based on the hybrid sampling-based measurement solution validates both approaches. The quality of data collected using each measurement approach for iteration-instrumented executions was found to depend on the application-specific measurement dilation of the given measurement technique. System noise was evident in both types of measurement, however, additional noise introduced in hybrid sampling measurements of parallel executions made their interpretation more difficult.

The techniques developed in this thesis and prototyped in the Scalasca toolset provided valuable insight into the performance dynamics of parallel HPC applications, assisting in understanding and tuning execution performance, as demonstrated by our real-world application case studies, conducted in cooperation with the application developers. Our analysis of *PEPC* on JUGENE contributed to the developers' understanding of a serious communication imbalance, enabling them to remove a significant bottleneck from *PEPC*, while our hybrid sampling-based measurement method has proven to be the first measurement technique capable of collecting high-quality, low-overhead call-path profile measurements of *DROPS*, giving the developers a new, powerful tool for analyzing the performance of their code.

Our main contributions can be summarized as follows:

- Appropriate source-code iteration instrumentation and presentation techniques that open up a new perspective on the performance dynamics of parallel applications.
- An on-line compression algorithm that reduces both run-time memory overhead and analysis report writing/storage costs, taking advantage of the inherent redundancy in measurement results collected using iteration instrumentation.
- A hybrid measurement method that provides a seamless combination of direct instrumentation for MPI communication events and sampling for computation, providing an alternative technique for low-overhead measurements in cases where compiler-instrumentation overhead would be excessive, while still collecting enough information for the compression algorithm to function properly.

Table 8.1: *Summary of measurement dilation and iteration analysis compression for direct instrumentation and hybrid sampling approaches with the test application suite.*

	Dilation		Cluster count			
	direct	hybrid	direct	hybrid	Iterations	Remarks
121.pop2	1%	4%	64	128	440	
122.tachyon	47%	1%	—	—	—	
125.RAxML	1%	0%	64	64	21 268	
126.lammps	2%	1%	64	64	300	
128.GAPgeofem	4%	3%	64	64	2 501	
129.tera_tf	26%	1%	64	128	190	
132.zeusmp2	1%	3%	16	32	200	
137.lu	1%	3%	64	64	180	
142.dmilc	4%	*1%*	—	—	8+241	*Some MPI ignored*
143.dleslie	2%	14%	64	256	15 054	
145.lGemsFDTD	0%	1%	32	64	1 500	
147.l2wrf2	15%	*2%*	32	64	720	*Some MPI ignored*
DROPS	204%	13%	N/A	32	250	
PEPC	5%	*N/A*	256	N/A	1 300	*No BG implementation*

- Analysis of the newly developed techniques using the SPEC MPI 2007v2.0 benchmark suite, and two detailed case studies of real-world applications, conducted in cooperation with the developers of the *PEPC* and *DROPS* codes.

Table 8.1 shows an overview of our experiences from applying the different techniques to the test application suite. The first two columns compare the measurement dilation (without compression applied) for the applications using direct instrumentation and the hybrid sampling method. We used the green color for dilation under 5%, yellow for moderate overhead (5%-15%) and red beyond 15%. The data shows that while direct instrumentation works very well in the majority of the cases, it introduces unacceptable levels of measurement dilation in a few cases. At the same time, the hybrid sampling solution provides reasonable overhead in every case, with only a few borderline cases. Also, taking a measurement using the hybrid sampling solution is easier as it does not require setting up a function filter list, as is generally the case with direct instrumentation. It is worth mentioning though that for *142.dmilc* and *147.l2wrf2* some MPI groups had to be turned off to get a low-overhead measurement, so in these cases some manual labor is necessary also when using the hybrid sampling-based measurement method.

Looking at the cluster counts of the two measurement methods, 64 proved to be the general sweet spot in the direct instrumentation case, while the hybrid sampling method generally needed around twice the number of clusters to provide a good representation, due to the increased amount of noise and less precise call-tree structure comparison.

8.1 Future work

It is clear that the work presented in this thesis is by no means "finished". We solved a number of questions and introduced new ones. We explored new areas and found new problems which could and potentially will fill several more theses. The goal was not to solve everything, as whenever we solve one problem, a dozen new questions arise. The goal was to create a solid foundation and build good solutions on it, paving the way for our future work. Part of this foundation work was the collection of extensive experience with a relatively large number of applications, a significant contribution to the understanding of the diverse characteristics and problems of HPC applications, especially focusing on time-dependent performance behavior. Still, in the near future we will extend the scope of applications studied even further, introducing OpenMP and hybrid MPI+OpenMP measurements combined with the techniques developed in this thesis.

A large amount of important implementation work also has to be done. We need to extend our support for the hybrid sampling capability to other architectures, in particular to Blue Gene, by reaching full integration of other third party unwinding tools beyond libunwind, especially *StackWalker API* [17], where some early results are already available in [84]. Another very important development area is reducing the impact of the compression algorithm on the application execution. One idea for this is to execute it in a synchronized way just after a collective synchronization (e.g. `MPI_Barrier`), which could be marked by a simple API, and adding another `MPI_Barrier` just after the compression has finished processing the current iteration. If this technique proves to be successful, it will open up a way of minimizing the performance impact of other potentially high-overhead activities of the measurement system, giving us more flexibility when designing new features for the measurement system.

There are still some unsolved questions, such as finding the optimal cluster count for a measurement, or how to tell when the cluster count is too low. An automated system with that capability would certainly be a useful asset, especially for less experienced users. While there are certain meaningful defaults and rules of thumb, and an experienced human can likely tell the difference between good and bad compression results, automating this process can still prove to be a significant challenge. One idea that works well in manual analysis is to compare the mean and the maximum curves of the iteration graphs. The mean usually has much better quality than the maximum, resulting in a smooth line for the mean and a jagged line for the maximum in cases where the cluster count was too low.

There is also ongoing work in co-operation with other members of the Scalasca developer team, where better integration of the iteration instrumentation data into the general Scalasca framework will become possible by introducing a new analysis file format [26], which is in the final stages of implementation at the time of this writing. Another unsolved problem that ties into this is developing better visualization techniques for large iteration and process counts and exploring new ways of visualizing the data, along with better integration of these techniques into the Scalasca analysis report explorer. In particular, we hope to be able to extend our studies in the very promising field of 3D visualization, providing a scalable solution for the visualization of time-dependent behavior integrated into our report explorer.

As for longer term goals, it is planned that these measurement techniques will be re-implemented in production quality, potentially reducing the abundance of features — a typical

characteristic of prototype implementations — to achieve smoother user experience. The re-implementation will take place in the framework of Score-P, a measurement system developed in cooperation between the developers of the performance tools Periscope [33], Scalasca, TAU and Vampir, in an effort to create a single, unified measurement infrastructure, shared by all of these tools. As part of this unified measurement system, the impact of this work will be greatly magnified, potentially evolving into a de facto standard in several areas, including low-overhead, small memory footprint measurement techniques and iteration instrumentation. With these contributions to the state-of-the-art of performance tools, we hope to provide HPC users with easy to use and powerful analysis techniques, assisting them in the quest of coping with the growing complexity of HPC systems as the world slowly approaches the era of exaflop supercomputers.

Appendices

Appendix A

Detailed Analysis of the Application Suite

In this appendix, we take a more detailed look at the basic characteristics of the test application suite. We present statistics about the suite's coverage of the MPI standard, first through an overview of the different MPI groups, and then breaking it down to individual MPI functions. After this, we compare the characteristics of both uninstrumented and instrumented measurements at a range of scales between 8 and 1,024 processor cores. As the rest of this work mainly focuses on 256-process measurements, it is important to understand how the discussed characteristics can be expected to change at different scales to get a full understanding of our overall results.

Even though the studied applications cover a wide variety of MPI patterns, it is interesting to point out that a large portion of MPI functions are still not used, as shown by Table A.1. The coverage of communication functions and those dealing with communicators, groups and topologies is quite good, while many of the less frequently used features, like parallel I/O or one-sided communication, introduced by version 2.0 of the MPI standard are not used. Still, the most important features are covered, and the set is expected to be representative of present day HPC applications using MPI as their means of communication. Note that the `CG` and `TOPO` groups are the ones where unwinding was disabled for two of the applications in Chapter 5 to reduce excessive unwinding overhead.

Tables A.2-A.4 tally the MPI functions used by 256-way benchmark executions (1024-way in the PEPC case). The values in the table are per-process average function call counts. Approximate values are shown where the average is not an integer. The table shows that a diverse range of MPI programming patterns are implemented, for example blocking, non-blocking, or persistent point-to-point communication, with or without extensive collective communication, etc. (SPEC rules allow only MPI parallelization, so auto-parallelization capabilities of compilers must be disabled, at least in the current versions of the benchmark suite.) The suite therefore provides a comprehensive test, both for MPI benchmarking purposes, but also for examining the effectiveness of parallel performance tools with real-world applications.

Table A.2 shows that except for the farming-based *122.tachyon* and *125.RAxML*, all applications show a wide variety and large number of point-to-point communication calls. Evidently, the most important, most frequently used function is`MPI_Irecv`, followed by `MPI_Isend` and `MPI_Send`. This is a good sign, as it means that a large fraction of the applications is

Table A.1: *Breakdown of the test application suite's MPI function usage by MPI function groups as defined in the Scalasca measurement system. The numbers suggest that only a small fraction of all available MPI functions is used in most cases.*

MPI group	Count	Used	Description
CG	25	8	Communicators and groups
CG_ERR	4	0	Error handlers for communicators and groups
CG_EXT	12	0	External interfaces for communicators and groups
CG_MISC	4	0	Misc. functions for communicators and groups
COLL	18	10	Collective communication and synchronization
ENV	7	3	Environmental management
ERR	9	0	Common error handlers
EXT	10	0	Common external interfaces
IO	52	0	Parallel I/O
IO_ERR	4	0	Error handlers for parallel I/O
IO_MISC	2	0	Misc. functions for parallel I/O
MISC	26	0	Misc. functions
P2P	34	17	Point-to-point communication
PERF	1	0	Profiling interface
RMA	14	0	One-sided communication
RMA_ERR	4	0	Error handlers for one-sided communication
RMA_EXT	7	0	External interfaces for one-sided communication
RMA_MISC	2	0	Misc. functions for one-sided communication
SPAWN	12	0	Process spawning
TOPO	20	5	Topology (Cartesian and graph) communicators
TYPE	35	0	Datatypes
TYPE_EXT	7	0	External interfaces for data types
TYPE_MISC	2	0	Misc. functions for data types
Sum	**311**	**43**	**Count over all MPI groups**

capable of using *asynchronous point-to-point communication* to overlap communication with computation, thereby reducing communication overhead.

The most frequently used collective communication & synchronization calls in Table A.3 are `MPI_Allreduce`, `MPI_Broadcast` and `MPI_Barrier`. `MPI_Reduce` is also used in more than half of the applications. The usage of collective communication is generally much more efficient than using self-written implementations of the same functionality based on point-to-point communication. Still, care must be taken when using these calls as many of them are *synchronizing*, which can easily become a scalability problem at large scales. `MPI_Reduce` and `MPI_Allreduce` are very powerful calls, providing an easy means to define calculations that summarize data in an efficient way from all processes in parallel. `MPI_Reduce` is most frequently used for collecting data on the master process for output, while `MPI_Allreduce` is more often used as part of intermediate computations, for example when synchronizing results between two iterations.

Table A.2: *Per-process call count of MPI point-to-point communication functions used by 256-way test application executions.*

	Irecv	Isend	Recv	Send	Ssend	Wait	Waitall	Waitany
104.milc	~11 486	~11 486				~22 973		
107.leslie3d	110 055			110 055		110 055		
113.GemsFDTD			~63		~63			
115.fds4			~841	~841			2 509	
121.pop2	~433 961	~433 961					649 672	
122.tachyon							1	
125.RAxML								
126.lammps	7 560		672	8 232		7 560		
127.wrf2	~243 383		~325	~243 708		~243 383		
128.GAPgeofem	~3 542 108	~3 542 108					610 626	
129.tera_tf	23 085		~5	~23 090		23 085		
130.socorro	~110 445			~110 445				~110 445
132.zeusmp2	33 010	33 010					7 809	
137.lu	~685		274 050	~274 735		~685		
142.dmilc	15 884	15 884				31 768		
143.dleslie	828 025			828 025		828 025		
145.lGemsFDTD	~36 914	~36 914	~66	~66			1 500	
147.l2wrf2	~105 338	~105 338				210 675		
PEPC	~667 772	~671 671	~3 899					~667 772
DROPS	~2339 791	~2 370 962	~34 409	~2	~11 658	~19 867	137 391	

	Recv_init	Send_init	Start	Startall	Test	Testsome	Probe	Sendrecv
113.GemsFDTD								~23 469
122.tachyon	~32	~32	~32	~0*		~0*		
DROPS					~26 351 997		~15 591	

Table A.3: *Call count of MPI collective communication functions used by 256-way test application executions.*

	Allgather	Allgatherv	Allreduce	Barrier	Bcast	Gather	Reduce	Scan
104.milc			~565	~2	~4			
107.leslie3d			4 401	258			2	
113.GemsFDTD				~5	~37 172		~4	
115.fds4			5 034	10		211		
121.pop2			68 978	724	980			
122.tachyon	1			1				
125.RAxML					59 128		8 628	
126.lammps			135	2	66			1
127.wrf2					2 109			
128.GAPgeofem			872 019		11			
129.tera_tf			382	1 445	37			
130.socorro	16	248	1 172		303		34	
132.zeusmp2			402	3	40		2	
137.lu			7	1	9			
142.dmilc			586	2	4			
143.dleslie			582	260			1	
145.lGemsFDTD				11	4 214		4	
147.l2wrf2			8		687	124		
DROPS	357	56	126 477	677	627	54		
PEPC	91 543	6 505	45 909	104 553	2	2 480	158	

	Gatherv	Scatterv
147.l2wrf2	62	62

Table A.4: *Per-process call count of miscellaneous MPI functions used by 256-way test application executions.*

	Comm_ create	Comm_ dup	Comm_ free	Comm_ group	Comm_ rank	Comm_ size	Comm_ split	Group_ range_incl
104.milc					~4 849 261	~22 709	1	
107.leslie3d					1	1		
113.GemsFDTD					1	1	256	
115.fds4					1	1		
121.pop2	3			3	3	3		3
122.tachyon					5	2		
125.RAxML					1	1		
126.lammps			1		8	5		
127.wrf2					4	4		
128.GAPgeofem		1			1	1		
129.tera_tf					1	1		
130.socorro		7			10	9		
132.zeusmp2					2	2	1	
137.lu					1	1	1	
142.dmilc					~20 113 719	24 893		
143.dleslie					1	1		
145.lGemsFDTD					1	1	1	
147.l2wrf2					2 300	127		
DROPS		56	140		197	197	112	
PEPC					1	1		

	Cart_ coords	Cart_ create	Cart_ get	Cart_ shift	Dims_ create	Request_ free
113.GemsFDTD					1*	
122.tachyon						~64
126.lammps		1	1	3		
132.zeusmp2	1	1		3		
147.l2wrf2	2	2		920 548	1*	
PEPC						~671 671

This is also reflected by the calling frequencies of the two functions. `MPI_Bcast` is in many cases used for distributing input data from the master process to the others, but it can also be used to send control messages to the slaves from the master, as is the case in *125.RAxML*. Performance experts often discourage frequent usage of `MPI_Barrier`, as it can very easily become a scalability bottleneck, often without a good reason. As the table shows, the developers are aware of this, and although most of them use `MPI_Barrier`, they use it sparingly. One exception is *PEPC*, where there is quite a large number of `MPI_Barrier` calls. This is because we were working with a development version of the code where they used these calls to separate the execution's main phases for their own profiling purposes. The current production version of *PEPC* does not contain `MPI_Barrier` calls in its main loop.

The functions in Table A.4 are mostly dealing with *communicator*, *group* and *topology* management. None of these applications uses these features extensively. We have seen problematic cases (at least for our measurement system) caused by the usage of extreme numbers of communicators in other applications [24], but this kind of problem is not represented in this suite. The most often used calls from Table A.4 are `MPI_Comm_rank` and `MPI_Comm_size`, two quite basic features of MPI. While most applications use these relatively rarely, many only once per process, *104.milc* and *142.dmilc* use them very often. Also, `MPI_Cart_shift` is used very often by *147.l2wrf2*. Interestingly, this is not the case in 127.wrf2, its predecessor. These MPI functions are typically less important from the parallel performance characterization point of view, but their frequent calls in some applications cause unnecessary, excessive measurement overhead in Chapter 5, where we deal with this problem separately.

A.1 Initial measurements

Figure A.1 shows a graph of the benchmark execution times with different numbers of processes, on a log–log scale, from which the scalability of each benchmark can be determined. Strong scaling applications appear as straight lines decreasing with larger process counts. To reduce the impact of variability in run times (due to non-dedicated use of the communication switch and filesystem in the production configuration of the JUROPA system), the best run time of several measurements is taken although this is contrary to the SPEC benchmark rules, where dedicated systems are used and the median of all run times is taken. Including confidence intervals in the graphs and tables would be appropriate in a comprehensive study, however, these have been omitted to reduce unnecessary clutter and clarify the underlying behavior. In general, measurements on the JUROPA system show significant run-to-run variation, which depending on the application and other circumstances can reach around 5%, while such variability on the JUGENE system is virtually non-existent, thanks to the better isolation of jobs on Blue Gene systems.

The SPEC MPI applications that do not provide a large reference dataset ('lref') were evaluated using the medium reference dataset ('mref'). These measurements were taken using 8-1024 processors. As the smallest configuration for the large reference dataset is 64 processes, the 8-32-way measurements were omitted in those cases. It was not possible to run *DROPS* using 1024 processes so 512 is the highest scale in that case. With respect to *PEPC*,

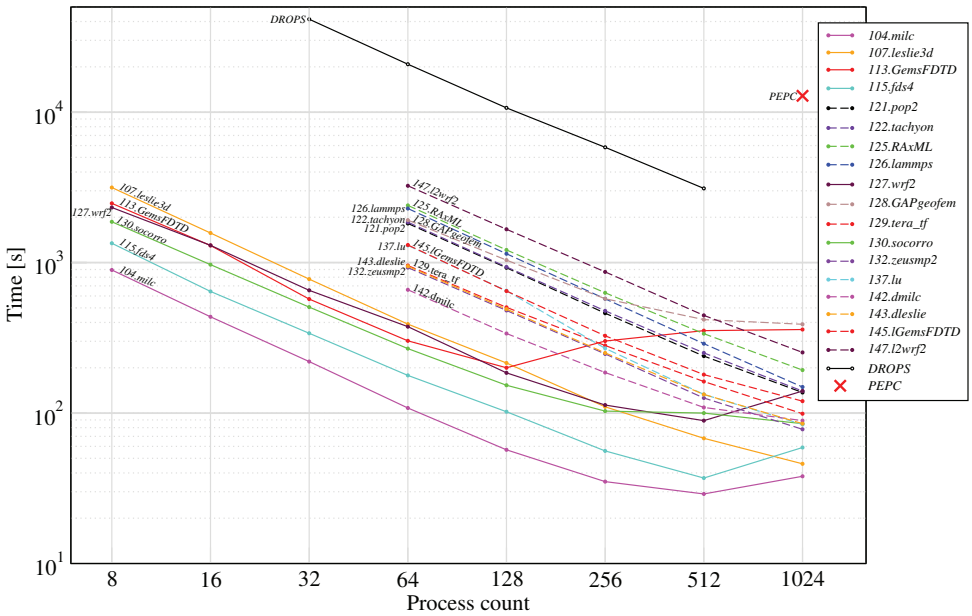


Figure A.1: *Application execution times at different scales.*

we were specifically interested in the 1024-way JUGENE configuration only, and the input data provided to us was configured for that scale.

While most of the benchmarks scale well, it is clear that certain others have very limited scalability, before no further speed-up is possible or performance degrades unacceptably. The most prominent example is *113.GemsFDTD*, where scaling beyond 128 processes causes a slowdown instead of a speedup, caused by the inefficient use of collective operations for distributing the input data. The scalability of this application was later improved based on our feedback to the original application authors [5]. *145.lGemsFDTD*, the SPEC MPI v2.0 version of the same application doesn't show the same scalability limitations. Other examples of bad scalability include *104.milc*, *115.fds4* and *127.wrf2* beyond 512 processes (all 'mref'-sized), and *128.GAPgeofem* beyond 256 processes, an 'lref'-sized measurement.

Although no tuning has been done for JUROPA, and measurements were taken on a non-dedicated production system, from review of published SPEC benchmark results [83] the same scalability limitations are seen to be comparable to specifically optimized benchmark measurements on dedicated systems.

Of course, analyzing the performance of optimally-tuned applications that scale perfectly has much less value than identifying potential opportunities for improvement of applications with problems, and is key to producing better performing and more scalable applications.

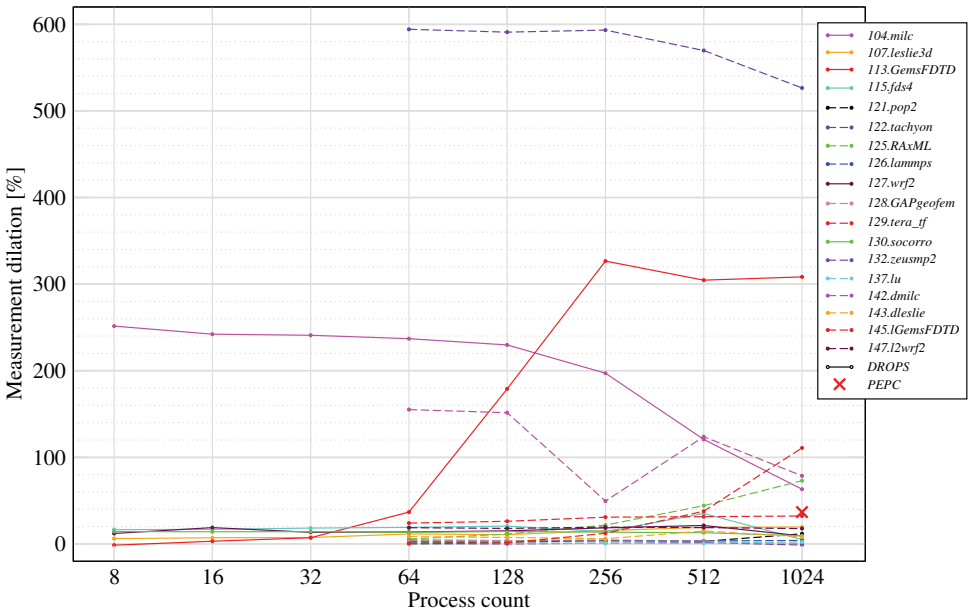


Figure A.2: *Measurement dilation percentage at different process counts, when using full compiler instrumentation without filtering.*

A.2 Basic Scalasca measurements

Figures A.2-A.4 expand upon the basic information shown in Figure 2.1 (page 21) by showing how overheads change with different process counts. Changes in measurement dilation are influenced by a number of factors, and as such are not easily predictable. In these cases perhaps the most important factor is that these test cases are strong scaling, solving the same problem size at every scale. This means less work to be done on every process, leading to shorter execution times, while the communication counts stay the same or even grow in some cases, depending on the application. These factors cause a larger frequency of measurement events both from user and MPI functions, leading to increased measurement dilation.

On the other hand, with growing process counts the ratio of time spent in MPI usually also grows, as we will see in Figure A.9. The additional MPI time is most often spent waiting in some synchronization or blocking communication calls as a result of inevitable workload imbalance. As in these time intervals actually nothing happens on those processes, there are also no measurement events, and so no measurement overhead in them, causing a decrease in the perceived measurement dilation. This illustrates that measurement dilation is actually non-uniform over time, depending on the frequency of measurement events in a given time interval. Characterizing this non-uniformity could be an interesting task, but generally reducing dilation to a more-or-less negligible level is clearly superior to that approach. Still, keeping these effects in mind never hurts the quality of our analysis.

Figure A.2 shows the measurement dilation at different scales, experienced when using full compiler-based instrumentation of every function call in the application source code. Around

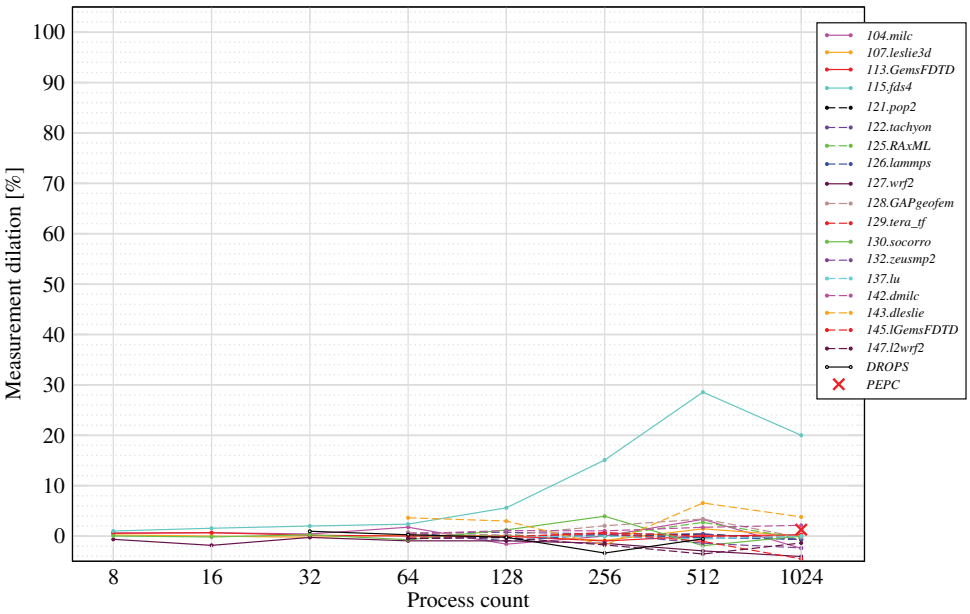


Figure A.3: *Measurement dilation percentage at different process counts, when using PMPI wrappers only.*

half of the applications show relatively stable overhead below the 20% line, while some of the others show extremely high, clearly unacceptable measurement dilation (e.g. around 600% for *122.tachyon*, and over 2000% for *DROPS*, which could not be shown on this scale). The correlation between process count and measurement dilation is not uniform. *104.milc* and *122.tachyon* show progressively decreasing overhead at higher process counts, whereas *113.GemsFDTD* and *145.lGemsFDTD* show increasing overheads.

Moving to the other end of the instrumentation-level spectrum, Figure A.3 shows measurement dilation when using only the PMPI wrappers for instrumentation. Note that the scale on the vertical axis is very different from that of Figure A.2. Clearly, in this instrumentation mode overheads are generally much lower, often in the 5% range, which is at the same scale as run-to-run variation.

By filtering out the most frequently called user functions we can reduce the overhead to acceptable levels in nearly every case, as seen in Figure A.4. Many applications have overheads in the 5% range, which is similar to the range of normal run-to-run variation, while most others have a value under the 15% threshold, with the prominent exceptions of *122.tachyon*, *129.tera_tf* and *DROPS* which still have excessive overheads. *DROPS* is not visible on the graph as its overhead is around 210% at all measured scales.

It is also interesting to note that some applications consistently show negative measurement dilation. Of course the measurement always causes positive amounts of overhead, but certain side effects can reduce or negate its impact. Specifically, our instrumentation disables certain optimizations (depending on the platform and compiler used), which may — in some cases — have a positive effect. We investigated this hypothesis using the *137.lu* application on JUROPA

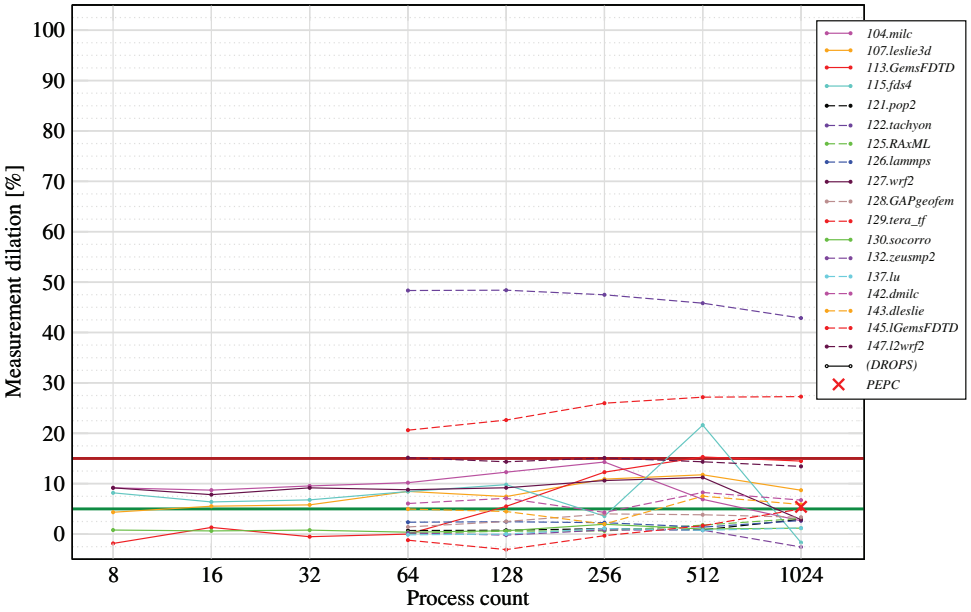


Figure A.4: *Measurement dilation percentage at different process counts, when using full compiler instrumentation with filtering.*

using the Intel 11.0 compilers, and found that disabling inter-procedure optimization actually improves performance. This also shows that great care has to be taken when interpreting performance measurement results of any kind. Measuring something is never possible without changing it. We can only try to understand and reduce the impact of this change.

Figure A.5 shows the same kind of overhead information when using the newly introduced hybrid sampling-based measurement method at 100Hz sampling rate. We see surprisingly heavy fluctuation and high overheads in a number of cases, nearly all of them being applications with mref-sized input datasets. The reason is that due to the strong-scaling nature of the input datasets, these applications have very little computation to do compared to the amount of communication (number of MPI calls). In this case, unwinding from every MPI call causes significant overhead compared to the relatively low computation time.

The solution to the problem is to use more reasonably sized, larger datasets, such as the applications with lref-sized cases. When looking at the larger datasets (marked with dashed lines) with more computation to do between communications, the overhead ratio descends under the 5% level for most of the applications, with some applications showing higher overhead at the 1024-process scale, again due to strong-scaling issues. The worst cases are *143.dleslie* and *DROPS* where the overhead is significantly higher than in the other cases and starts growing heavily at much lower scales than in the case of the other applications. The bottom line here is that measurement dilation is present, in some cases it even reaches the 15% threshold, but there is no case with extreme overhead (hundreds of percents), and most applications show less than 5% overhead when using large enough input cases.

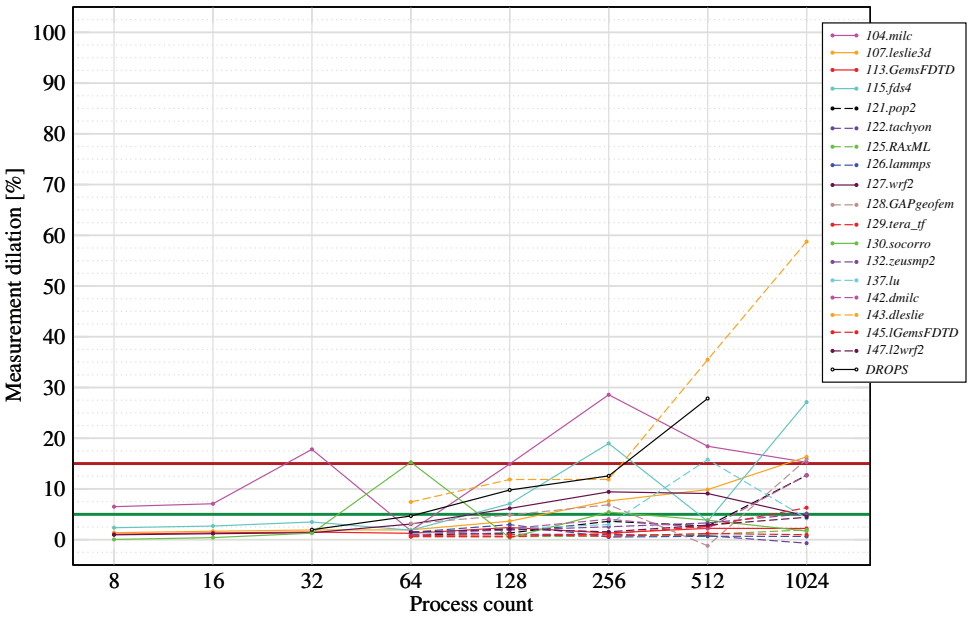


Figure A.5: *Measurement dilation percentage at different process counts, when using the hybrid sampling method at 100Hz sampling rate.*

Figure A.6 shows the function call and MPI event frequency when using full compiler instrumentation without filtering at different scales. Fig A.7 shows the same data when using filtering. It is clear from comparing these two figures that filtering has great potential in reducing the event count, often by several orders of magnitude. Comparing the filtered graph to Figure A.8, which shows the MPI call frequency, we see no great differences. This means that for most applications filtering reduces the compiler event count to a negligible level compared to the MPI call frequency, since most of the measurement overhead comes from MPI events.

Looking at the data of *DROPS*, which had the most extreme overhead from compiler instrumentation, we see that it has an extremely high frequency of calls, as expected, but actually not the highest. Among the large reference sized SPEC MPI examples, both *122.tachyon* and *142.dmilc* feature higher values at some scales. Both of these applications show significant measurement dilation without filtering, but not as high as *DROPS*. Similarly, after applying filtering, *DROPS* is in the higher call frequency ranges, but certainly not the very first. One reason for it still having the highest overhead when filtering can be that it still has to check the filter at a very high frequency, which involves looking up the current function from a very long list of function names.

Another message to take away from these graphs is that the MPI call frequency grows significantly with the scale, due to the strong scaling nature of our input data sets. This is an important factor in the growth of the unwind overhead with scale, as seen in a number of applications in Figure A.5.

Figure A.9 shows a very important characteristic derived from our measurements: the ratio of time spent in communication and synchronization calls to the overall execution time. Ideally,

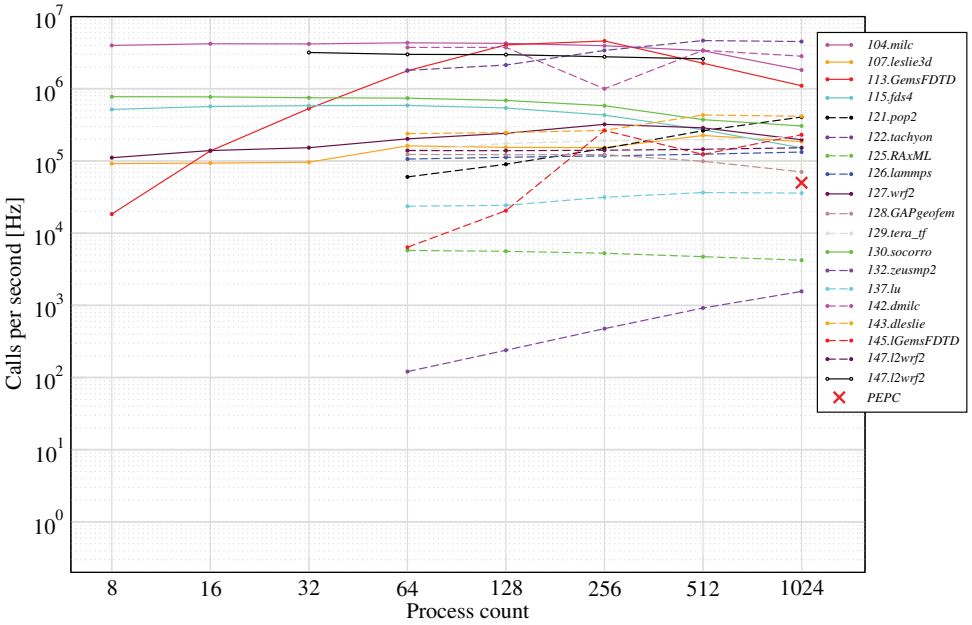


Figure A.6: *Function call and MPI event frequency when using full compiler instrumentation without filtering.*

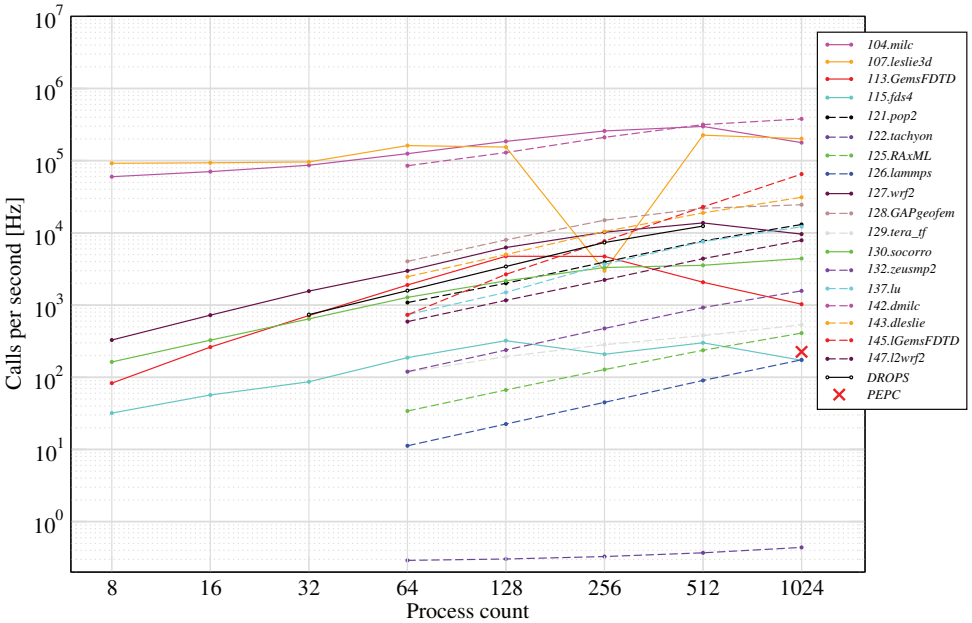


Figure A.7: *Function call and MPI event frequency when using full compiler instrumentation with filtering.*

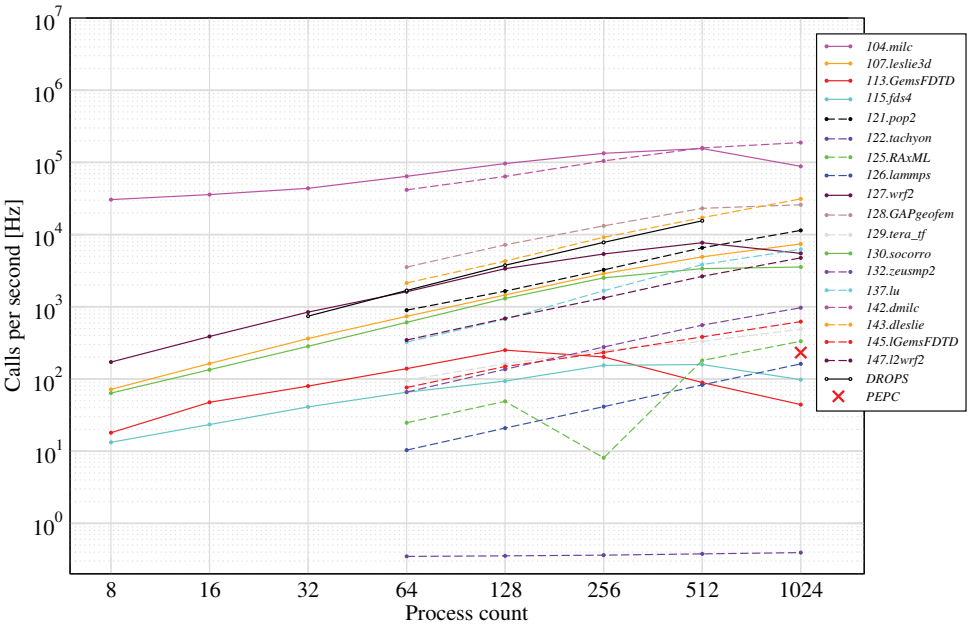


Figure A.8: *MPI call frequency.*

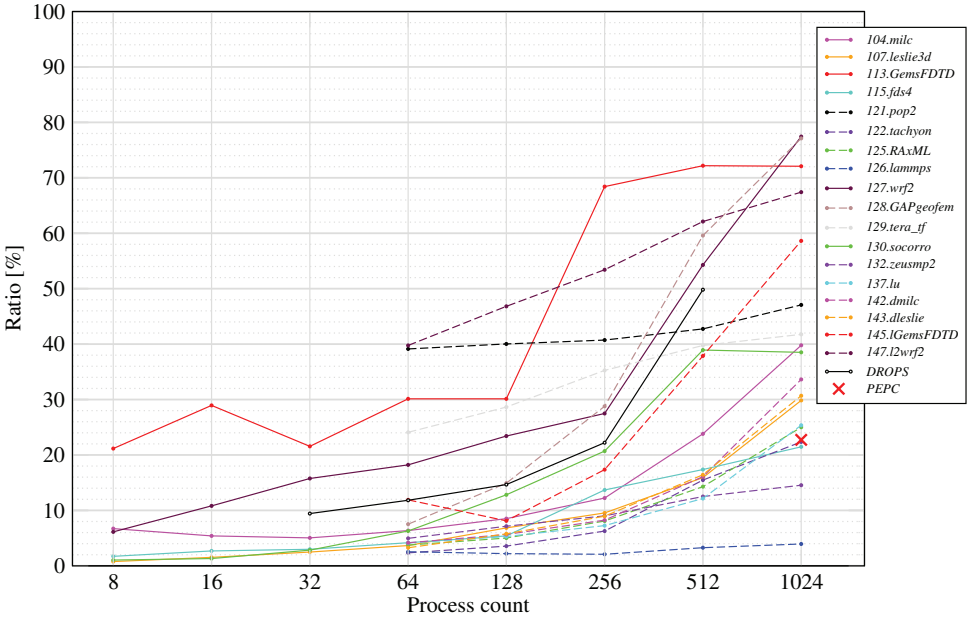


Figure A.9: *Percentage of overall execution time spent in MPI communication and synchronization at different process counts on the* JUROPA *system.*

no time would be spent communicating and we could use every cycle of every processor to do useful computations. In real life HPC applications, this is never the case. In fact, as the level of parallelism increases, time spent in communication tends to increase in all but the simplest cases, and it is generally only a matter of adding a few more orders of magnitude until communication time starts to dominate computation time. This is because in the performance model of any application, there are always some small, 'negligible' constituents which cause no harm at all at smaller scales, but easily ruin performance as the scale is raised to a sufficiently high level.

We are only considering moderate scales here by present day's standards, as the largest measurements in the graph are 1024-way, but these effects are already clearly illustrated in this graph. In some cases slowly, in others more rapidly, the proportion of time spent in MPI grows in every case. While the percentage is under 10% in nearly all 64-way measurements, most applications fall into the 20-50% range at 1024 processes, with some as high as 75%. 75% means that only 1 out of 4 processor cycles on average is spent on useful computations. This is a huge waste of resources, a clear scalability bottleneck, and reducing the impact of these problems on HPC applications is the main goal of performance analysis tools. The usage of these tools raises the user's awareness of the scalability problems, and focuses the optimization effort on the most important bottlenecks.

Appendix B

Overview of the Application Suite's Time-dependent Behavior

Figures B.1–B.16 show graphs and value maps of a selected set of metrics for each test suite application to illustrate the breadth of time-dependent performance characteristics found in our test application suite using 256-process executions on JUROPA.

The graph in the upper left in each set is slightly different from the other graphs. It still shows the iterations on the horizontal axis and metric values on the vertical axis, but it is a composite graph that gives a compact overview of the amount of time spent in computation and communication in each iteration. It shows *Computation time* (light blue), *Point-to-point communication time* (magenta), *Collective communication time* (violet) and *Collective synchronization time* (black) stacked on top of each other. All these values are averages of all 256 processes. Ideally, *Computation time* should cover an overwhelming proportion of the graph, and all others should be negligibly small, but as we have seen in our earlier analysis, this is only rarely the case in real-life applications, and the situation becomes progressively worse, as we scale applications to higher process counts. Note that collective synchronization time in `MPI_Barrier`s is negligible and only visible at all for *129.tera_tf* & *PEPC*, this is why the black color seems to be absent in most of the cases.

The rest of the graphs in the first column are back to "normal", as described in section 3.1. As opposed to the top left graph, these graphs are not stacked, all their values should be interpreted as the distance between the top of a colored bar and the horizontal axis.

The value maps on the right-hand side show the same metrics as the graphs on the left do, just in a different view. They show the process ranks on the vertical axis, and represent the values using the darkness of the colors on the value map. These value maps provide a better understanding of the distribution of values both in space (processes) and time (iterations), while the graphs on the left provide a clear understanding of the proportions of the values. The value map in the top right always shows *Exclusive execution time*, which corresponds to the light blue part of the overview graph on its left.

B.1 121.pop2

The first thing that comes to mind looking at the *Execution time overview* graph is that quite a large fraction of the time is spent in MPI throughout the execution. Around 60% is spent with calculations, while 10% is in point-to-point and 30% in collective communication. The overall baseline behavior does not change much over time: after an initial period of slightly decreasing iteration time the baseline remains relatively constant until the final iteration. Looking at the *Communication count* graphs, it is clear that these changes in the execution and communication times are driven by the fact that the amount of work done per iteration is changing over time. We can see remarkably good correlation here. Still, the most time is spent in collective communications by those processes that spend the least time in computations. This is a classic indicator of waiting time at a synchronization point: the processes that have less to calculate and/or communicate than the rest have to wait the longest for the other processes at every synchronizing collective. As we see in the *Communication count* graph, the difference between the lowest and the mean value is very large, while the differences between the processes in the *Communication time* graphs are also very large. This means serious communication imbalance, which is bound to ruin performance at higher scales. Also, the application spends more time in specific iterations at regular intervals, around every 17 iterations, likely because certain aspects of the model are only updated in these iterations.

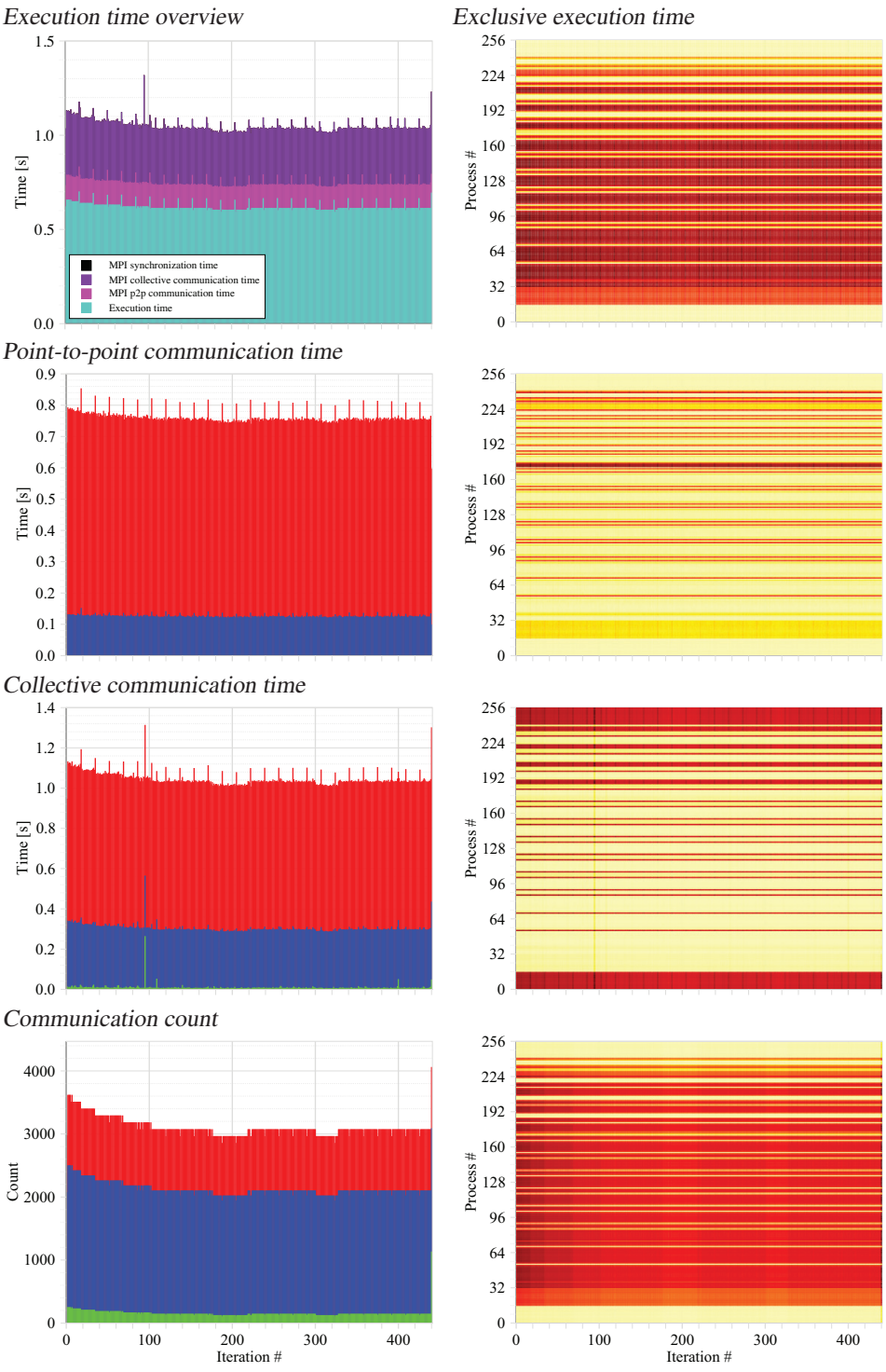


Figure B.1: *Iteration graphs and value maps of 121.pop2.*

B.2 125.RAxML

125.RAxML has over 20,000 iterations, so showing an overview and the fine details at the same time is not possible. This is why we provide two sets of graphs here, one for an overview of all iterations in this section, and one for showing some finer details for 500 iterations in the next section. This is a farming application in the sense that there is a master process that spends its time managing the workload of the others. The only communications are collective communications between the master and the rest of the processes. This is a highly efficient, very scalable communication scheme, what is known as *embarrassingly parallel*. As it is clear from the overview graph, an overwhelming proportion of the time is spent with useful calculations and communication time is negligible. Looking at the call trees of the iterations, it is also clear that this application uses a large variety of very different types of iterations. Looking at the corresponding source code we found that in every iteration a switch statement chooses the action executed in this iteration from a long list of possibilities, based on the input from the master process. This variability can be observed on virtually all the graphs, as the iteration execution characteristics switch back and forth between well-defined states.

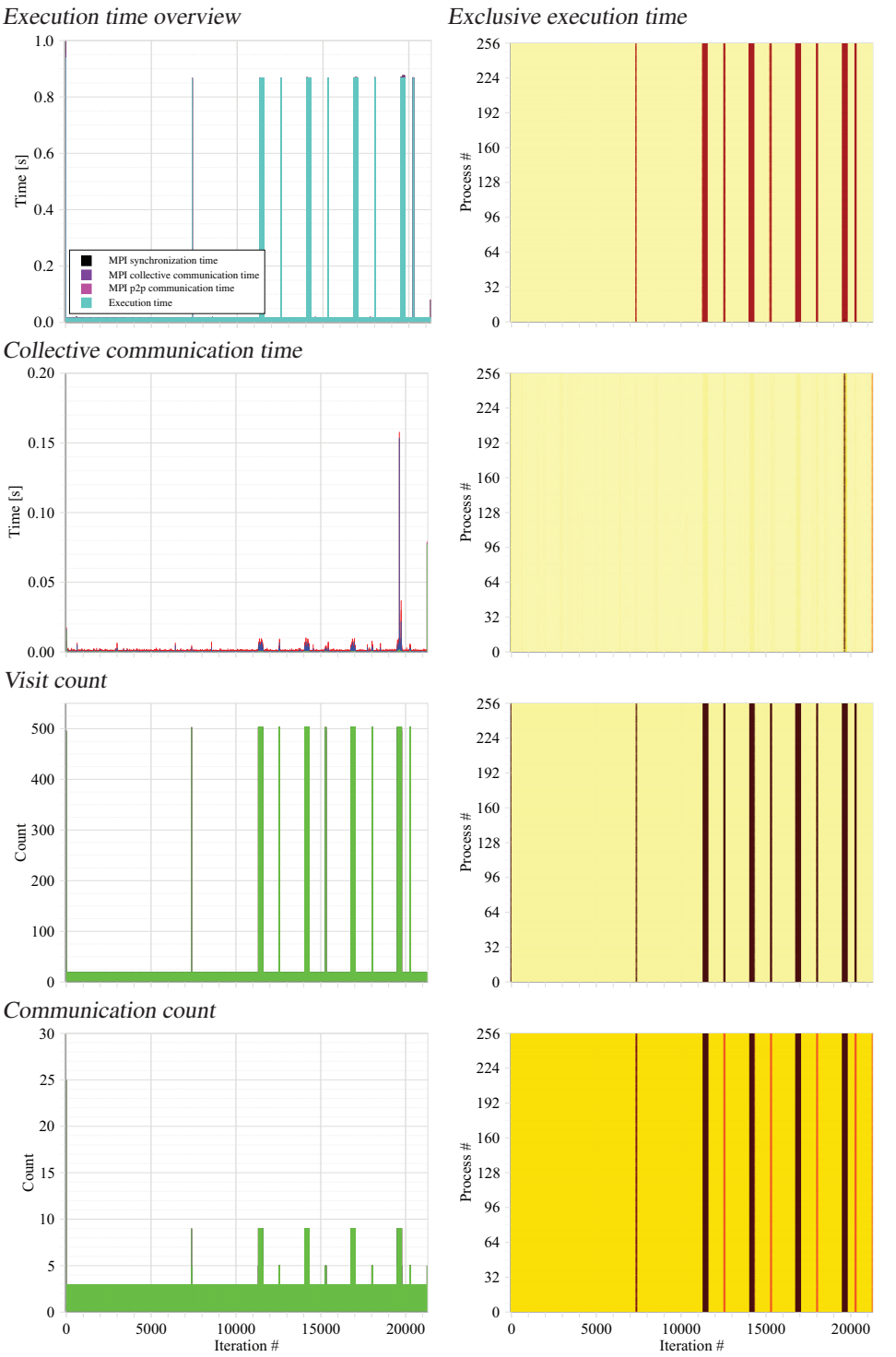


Figure B.2: *Iteration graphs and value maps of 125.RAxML.*

B.3 125.RAxML zoom

Zooming in on one of the peaks around iteration 11,500 we see that there is some very systematic, fine-grained behavior in such "peaks", as shown by Figure B.3. These are not simply a few iterations that take much longer than others. This is a range of iterations where the application keeps switching between different iteration modes, but not the same ones as before.

While there are many interesting features to study here, this is not a very interesting application from a parallel performance analyst's point of view, as there is nearly no time spent in communication. Our measurements show that the communication in this application is effective and uses the processor resources efficiently.

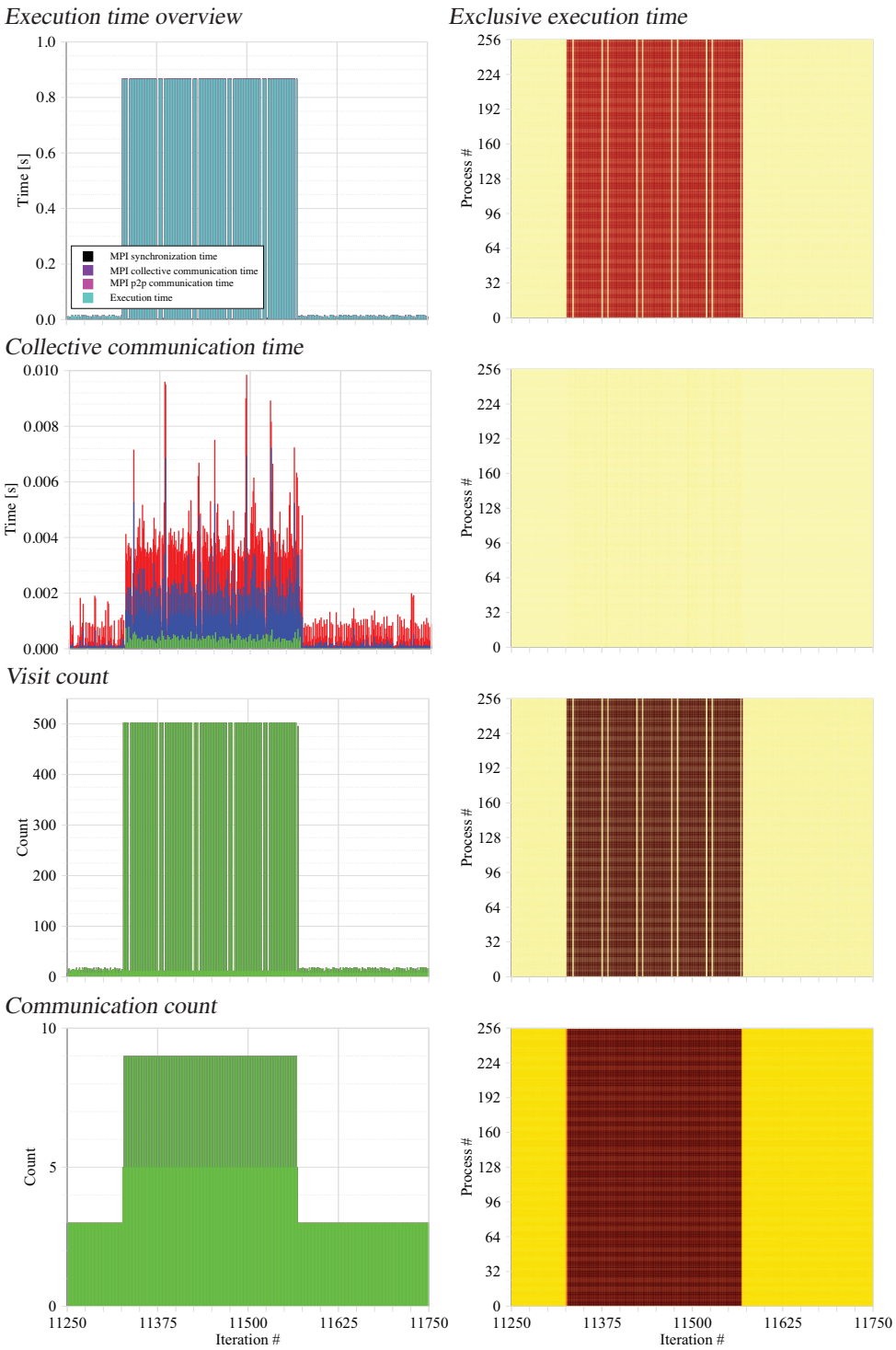


Figure B.3: *Iteration graphs and value maps of 125.RAxML (zoom).*

B.4 126.lammps

126.lammps spends most of its time in useful computations, so this is also not an extremely interesting example at this scale. It has a flat baseline behavior with systematic peaks every 5-8 iterations, at irregular intervals. As seen on the *Visit count* graph, the peaks are caused by more work being done in those iterations. The point-to-point and collective communication time metrics also show good correlation with these peaks in the *Visit count*, but they are still negligible at this scale.

There is also a regular pattern observable in the value maps of the point-to-point and collective communication time metrics. *126.lammps* is another of the SPEC MPI 2007 applications using a *Cartesian topology*. Looking at the topology display in the Scalasca GUI reveals a checkerboard-like pattern in the 3-dimensional Cartesian topology, a characteristic of the way communication is organized in *126.lammps*.

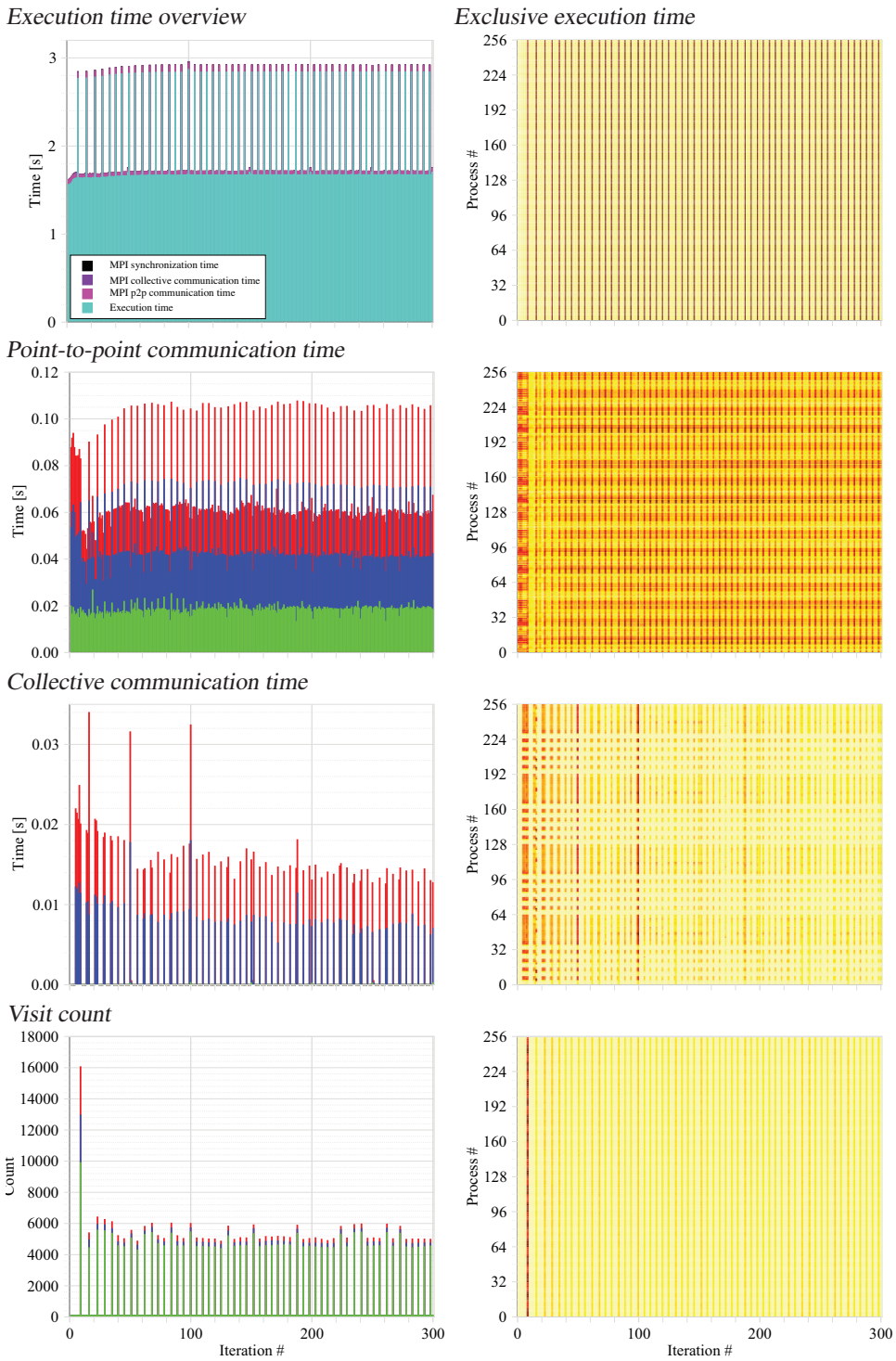


Figure B.4: *Iteration graphs and value maps of 126.lammps.*

B.5 128.GAPgeofem

128.GAPgeofem introduces a new kind of time-dependent behavior, sudden change in the baseline. This shows why it is important to analyze a longer range of iterations: if we only measured the first 500 iterations, we would think that this application has nearly perfectly flat behavior, and would not realize the sudden change just a few iterations later. The change is reproducible, also appearing in the *Visit count* metric which is a deterministic metric. A significant amount of time is spent in communications, so there is room for potential improvement here. Roughly the same amount of time is spent in point-to-point and collective communications, which show significant variations among processes. It is also typical that *Collective communication time* shows up as the inverse of *Point-to-point communication time*: where less time is spent in one, more time is spent in the other, caused by the use of synchronizing collectives. This latter behavior is somewhat harder to see on the standard maps, as it involves relatively small changes in the baseline. We have the capability to generate *histogram-equalized* value maps in order to maximize the contrast and make such small-scale details more visible. These value maps for the appropriate communication metrics are shown in Figure B.5.

Although some noise is present throughout the execution, more pronounced noise is found in the *Communication time* for several of the later iterations.

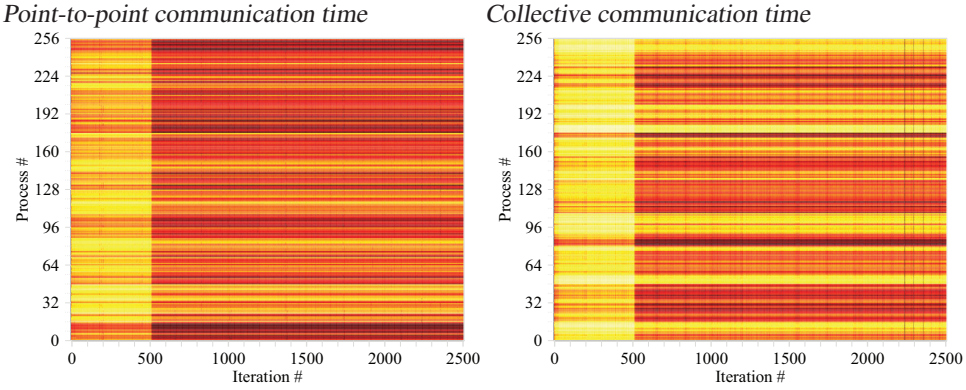


Figure B.5: *Histogram-equalized value maps of communication time metrics in 128.GAPgeofem.*

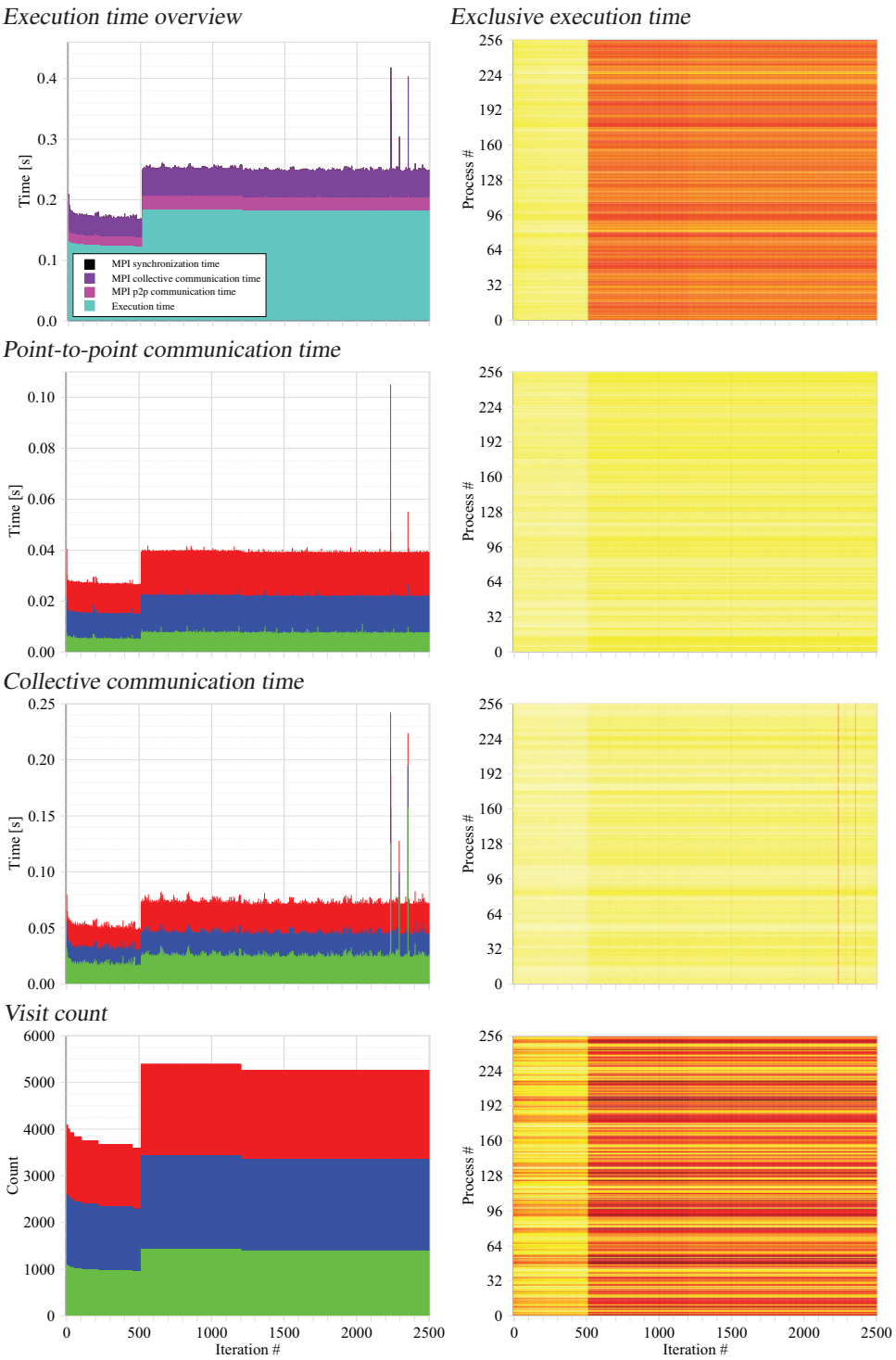


Figure B.6: *Iteration graphs and value maps of 128.GAPgeofem.*

B.6 129.tera_tf

This is one of the more interesting cases. The *Execution time overview* graph shows that *Exclusive execution time* is roughly constant over time, while *Communication time* shows significant gradual increase. In the first iterations, *Communication time* is still a small portion of the *Inclusive execution time*, which grows to around 50% by iteration 180. While *Point-to-point communication* dominates it throughout the execution, *Collective communication time* also grows from a quite insignificant to a significant level. What makes the problem more puzzling is that the *Bytes transferred* metric (and counts of communication operations that aren't shown) is constant throughout the execution.

Looking at the *Exclusive execution time* value map gives a more detailed impression of what is happening. The *Execution time overview* graph, showing only the average values, was misleading. *Exclusive execution time* is actually not constant. It is nearly constant on average, and it is constant on most processes, but a number of processes show the gradually growing behavior that we observed in the communication metrics. This suggests that the root of the problem is that those processes have some additional processing to do, and the time required for this grows over time. This leads to waiting times in Point-to-point communications on the processes which do their work in constant time, and the *Execution time* is finally evened out by a synchronizing collective call.

It is important to note the similarity between the graphs and value maps of *129.tera_tf* and *132.zeusmp2* (Figure B.8). As we have seen previously in Figure 3.1(a), the very characteristic growing stripes in the *Exclusive execution time* value map of *132.zeusmp2* are caused by higher workload in the middle of a three-dimensional topology. We can not directly access 129.tera_tf's logical topology, as it does not use `MPI_Cart_create` to create one, but we know that a similar topology is logically present in the code from the application's description on the SPEC MPI 2007 website [83].

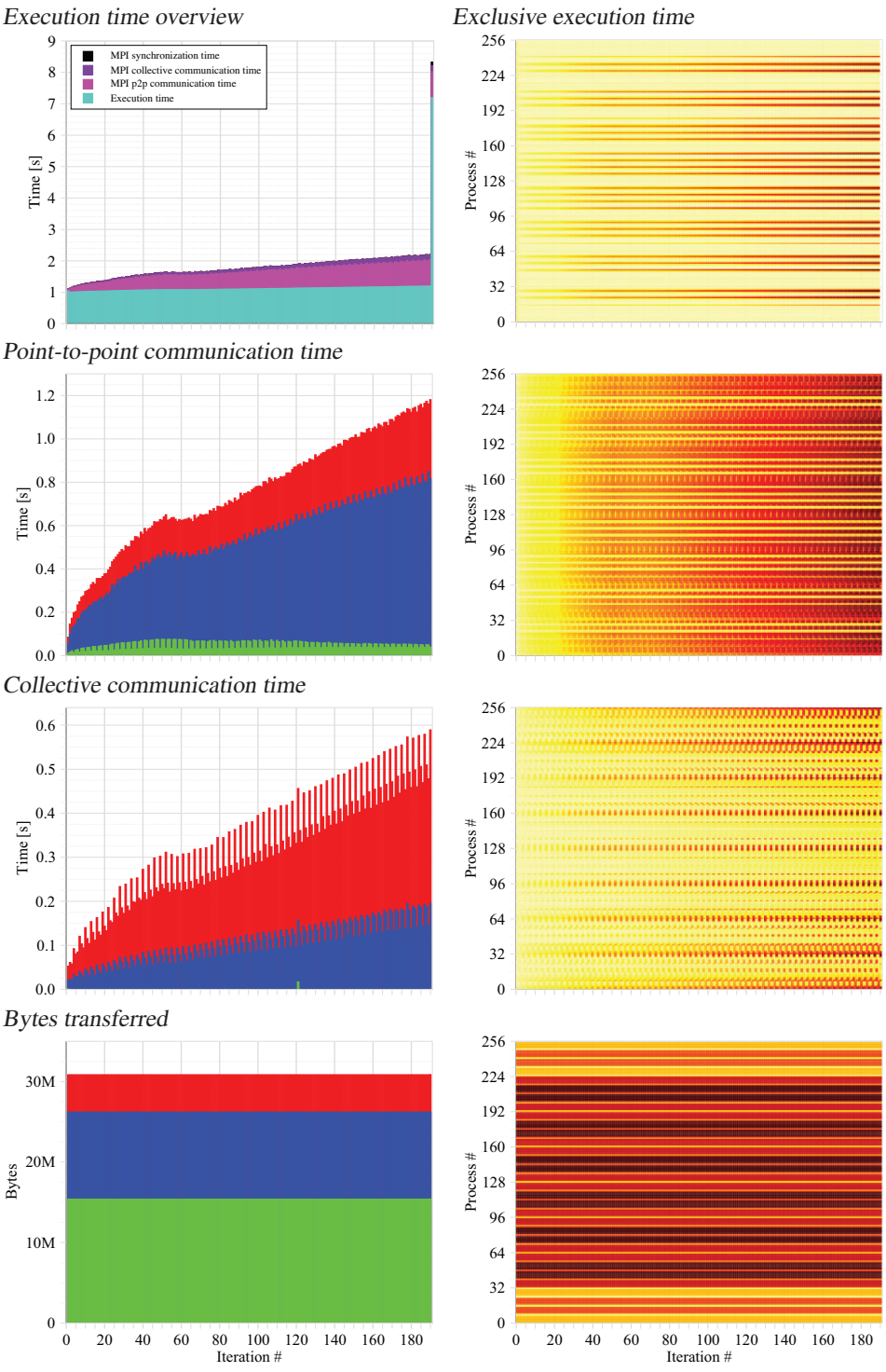


Figure B.7: *Iteration graphs and value maps of 129.tera_tf.*

B.7 132.zeusmp2

As pointed out in the previous subsection, *132.zeusmp2* is similar to *129.tera_tf*, showing a constant average *Exclusive execution time* with gradually growing *Communication time*, of which *Collective communication time* is the more dominant here. The *Bytes transferred* is constant over time again, and the root of the problem seems again to be located in the gradual increase of the *Exclusive execution time* on a subset of the processes, which are known to correspond to the middle of the logical topology in this case [10]. Nearly everything we said about *129.tera_tf* is true for *132.zeusmp2* as well. Maybe the most important difference is that the first 32 processes of *132.zeusmp2* show communication patterns quite different from the rest of the processes. There seems to be no reason for these differences in the *Exclusive execution time* or the count-based metrics like *Bytes transferred*, so this difference is unexplained.

Still, the remarkable similarities between the performance characteristics of the two applications suggest that this kind of behavior is not a one-off, special case, but rather the rule for a certain class of applications, and as such is an important candidate for further studies.

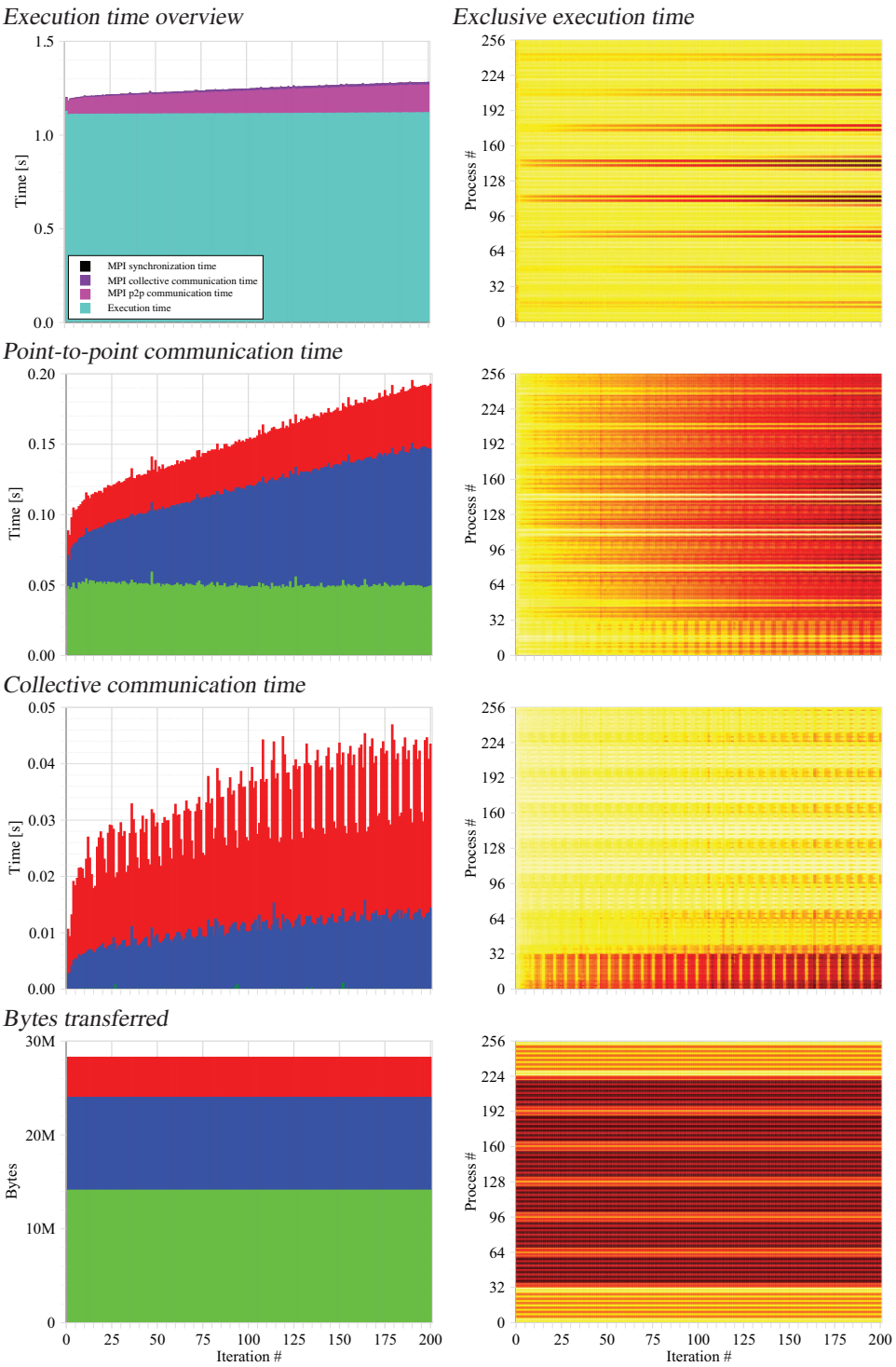


Figure B.8: *Iteration graphs and value maps of 132.zeusmp2.*

B.8 137.lu

The interesting thing about *137.lu* is that its main loop does not have any collective synchronization points, so the processes are just loosely synchronized by point-to-point communications. The behavior is basically just a flat baseline with constant amount of work done in each iteration, as illustrated by the *Visit count* graph, with an overwhelming amount of time spent in useful computation and the rest in point-to-point communication and a single synchronizing collective communication in the very last iteration. The periodic peaks in the point-to-point communication seem to be the result of such peaks showing up on ranks 48-55 in the *Exclusive execution time* value maps. Indeed, the *Point-to-point communication time* value map shows the peaks on every rank except for these, which is a good indication of the correlation. It is also interesting to note that these peaks are also visible in the *Inclusive execution time* value map in Figure 3.2 on page 34, where we also see that the peak in the *Exclusive execution time* of one iteration causes a peak in the *Point-to-point time* of the *next* iteration. This is caused by the propagation of the computational workload imbalance into the subsequent iteration, made possible by the lack of collective synchronizations in the iterations. The computational imbalance leads to waiting times in the point-to-point communications.

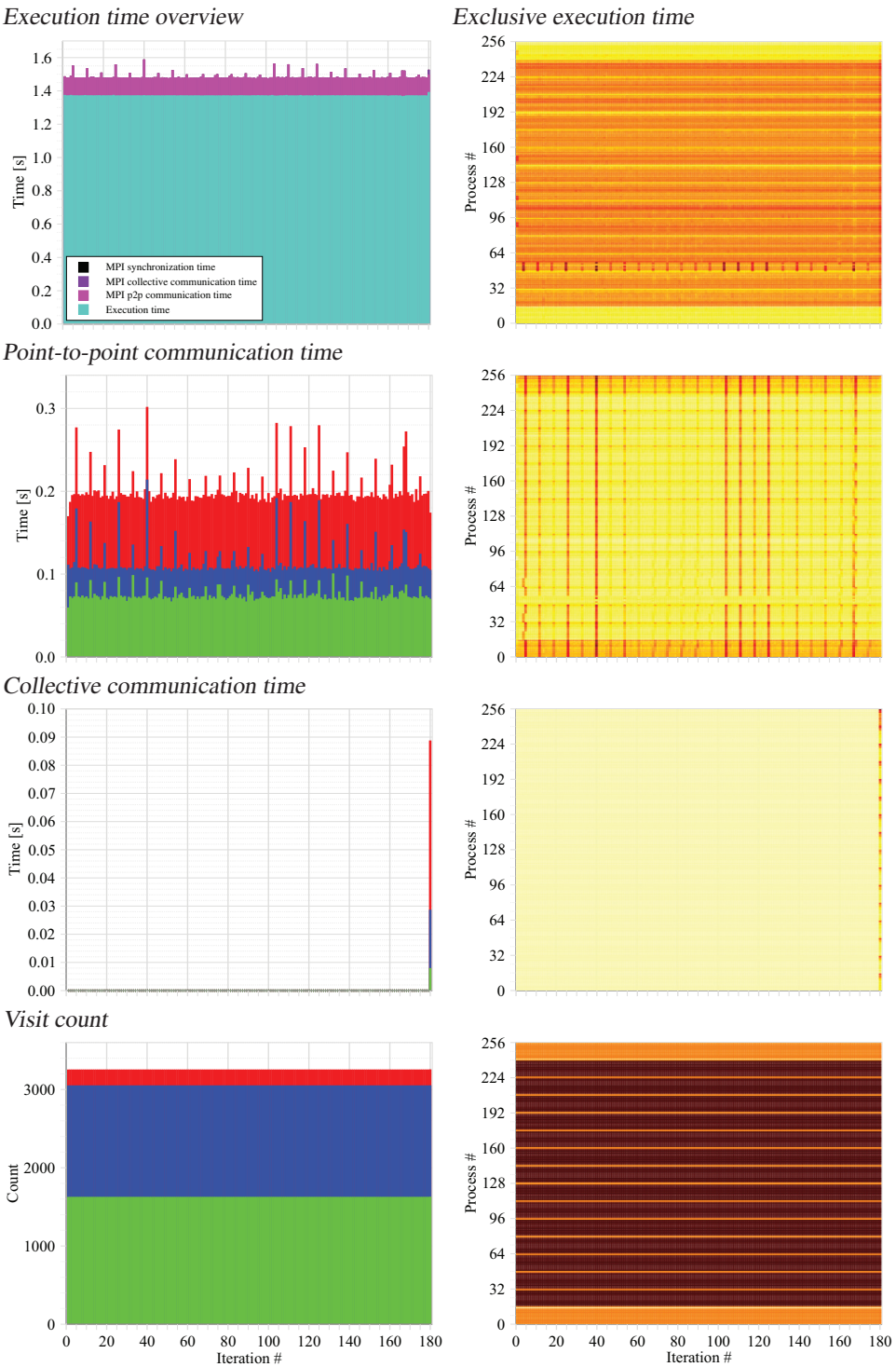


Figure B.9: *Iteration graphs and value maps of 137.lu.*

B.9 142.dmilc

142.dmilc is a complicated application with many loops, but no clear main loop. We instrumented the conjugate gradient loop, where around 20% of the execution time is spent. Most of the time in the loop is spent in actual computations, with around 10% spent in *Point-to-point communication time*. There are no clear changes visible over time other than the system noise, which can be expected from such a simple kernel. The *Communication count* graph shows that there are also not many communication events in the iteration loops, and their count stays constant over time. These communication events are overwhelmingly point-to-point communications. Overall, this loop of the *142.dmilc* application is relatively uninteresting from the performance dynamics point of view.

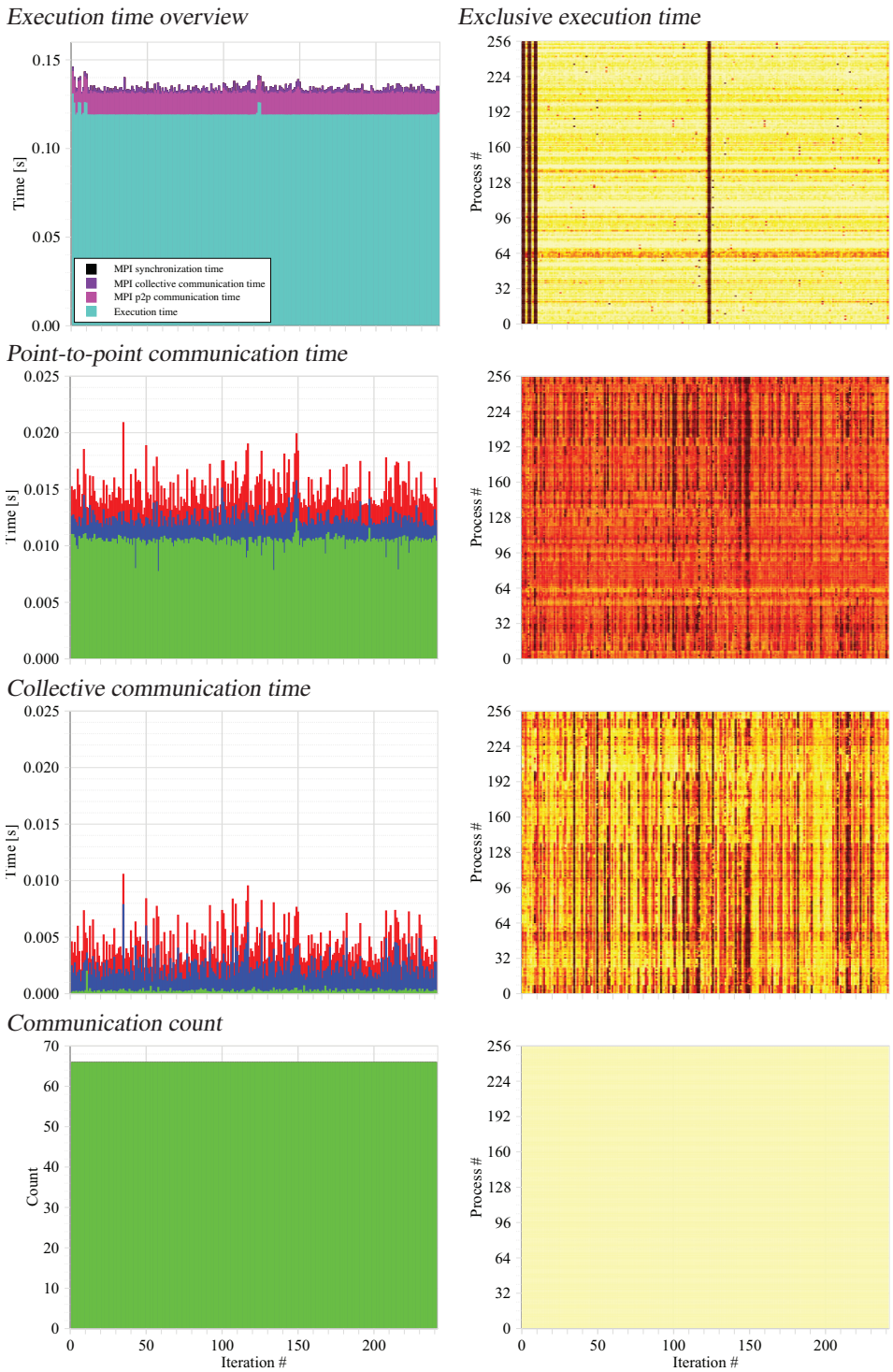


Figure B.10: *Iteration graphs and value maps of 142.dmilc.*

B.10 143.dleslie

143.dleslie is an application with lots of short iterations, so this is another case where we decided to present a set of graphs for both the overview and a subset of the iterations for more details. The *Execution time overview* graph indicates that most of the time is spent in useful computations. The *Exclusive execution time* shows a gradually increasing baseline with periodic spikes. The communication is overwhelmingly point-to-point with periodically appearing collective communication. The collectives seem to coincide with the peaks in the *Exclusive execution time* suggesting that the same action, taken at periodic intervals causes both of them.

Significant amounts of noise in the *Communication time* seem apparent throughout the execution's series of very short iterations.

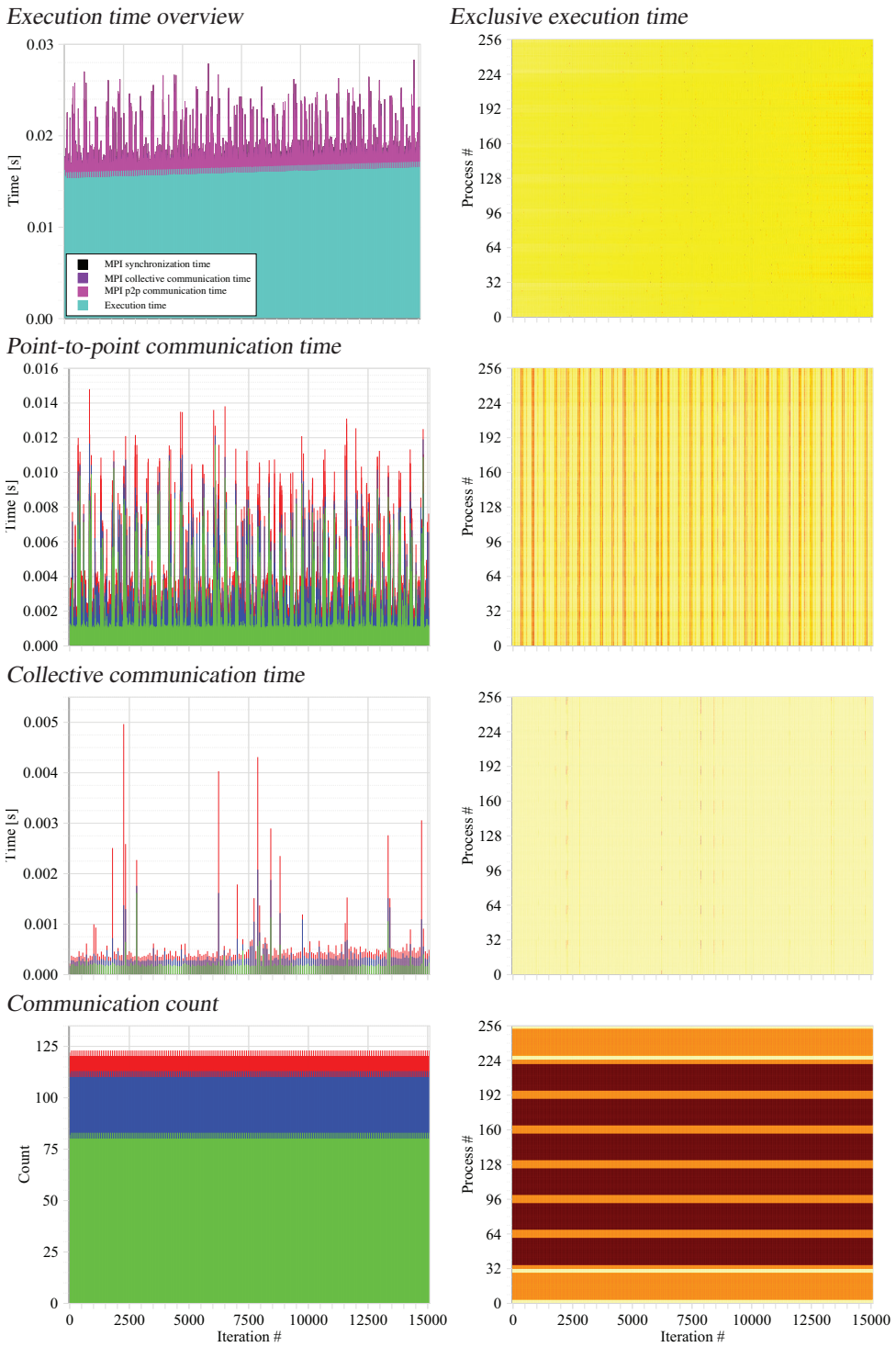


Figure B.11: *Iteration graphs and value maps of 143.dleslie.*

B.11 143.dleslie zoom

Interestingly enough, the peaks of the *Point-to-point time* seem to be relatively uncorrelated with the much smaller peaks in *Exclusive execution time*. At first glance these peaks might seem to be simple noise, but zooming in on a shorter range of iterations shows that this is indeed quite organized, systematic behavior. Around every 400 iterations there is a double peak in the *Point-to-point communication time*, the two peaks separated by around a hundred iterations. These peaks are not individual iterations, but each consist of around twenty iterations themselves.

It is obvious that there is some highly sophisticated behavior present here which would require additional analysis for complete understanding. However, as most of the time is spent in useful computations at this scale, *143.dleslie* is not the highest priority candidate for further investigation into its communication performance.

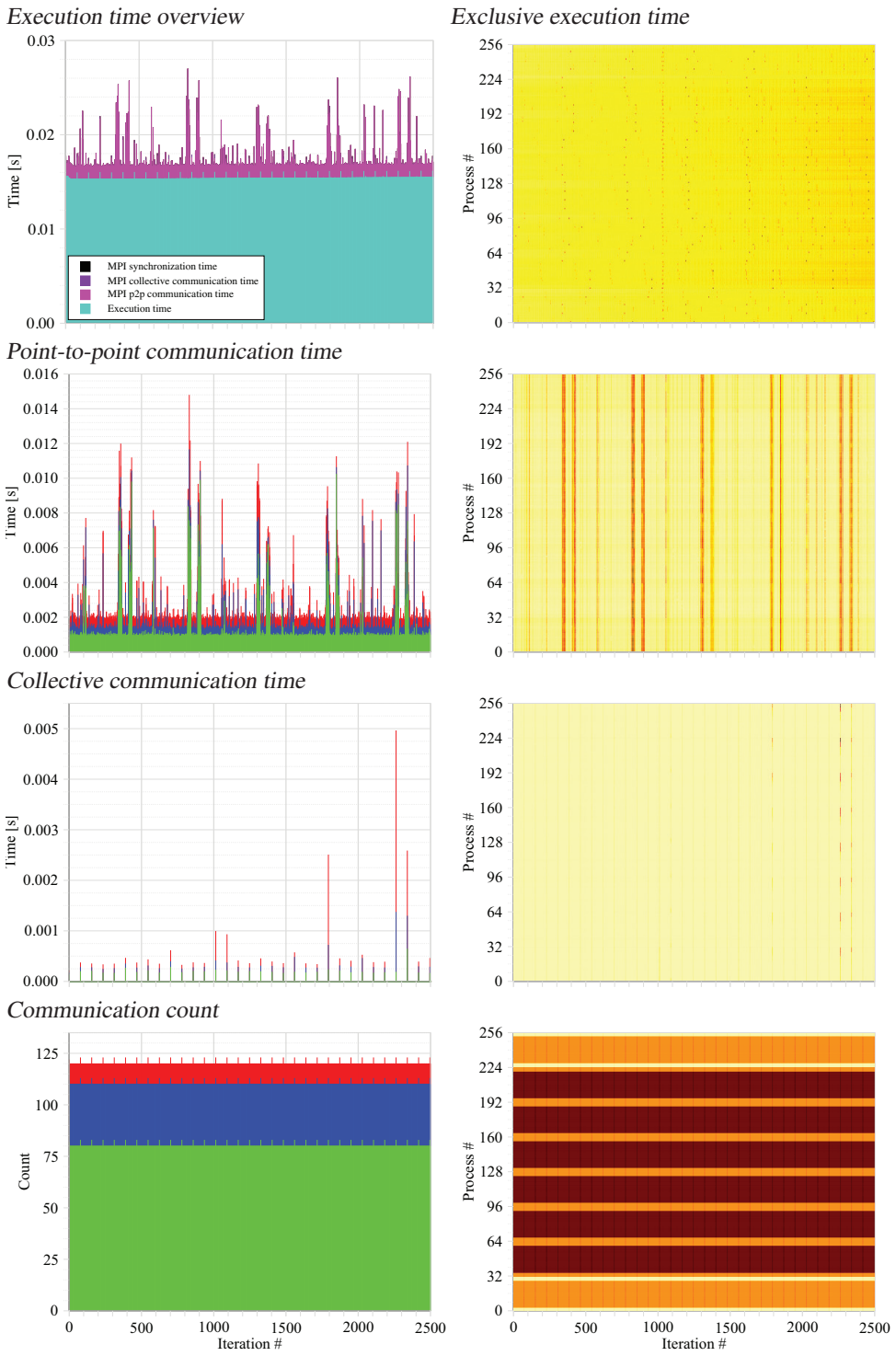


Figure B.12: *Iteration graphs and value maps of 143.dleslie (zoom).*

B.12 145.lGemsFDTD

Just by looking at the *Execution time overview* graph, *145.lGemsFDTD* seems to be a rather uninteresting application with a slight increase in the baseline *execution time* over time and around 10% of the time spent in point-to-point communication, and with collective communication showing up only on the first iteration. Drilling down a bit deeper, we find that there is an unusual pattern of workload imbalance in the *Bytes transferred* metric, with more data being communicated on the top 144 process ranks. The quite significant imbalance in *Point-to-point communication time* is not correlated with the pattern seen in the *Bytes transferred* metric. Rather, it is the inverse of the pattern found in the *Exclusive execution time* value map, suggesting that the imbalance is caused by actual imbalance in the computational workload. From our prior experience with *145.lGemsFDTD* [5] we know that this is an *MPMD (Multiple Program Multiple Data)* code, meaning that not all processes are doing the same kind of computation, separate process groups are assigned to do different kinds of calculations. This separation of responsibilities between processes is reflected well in our measurement results.

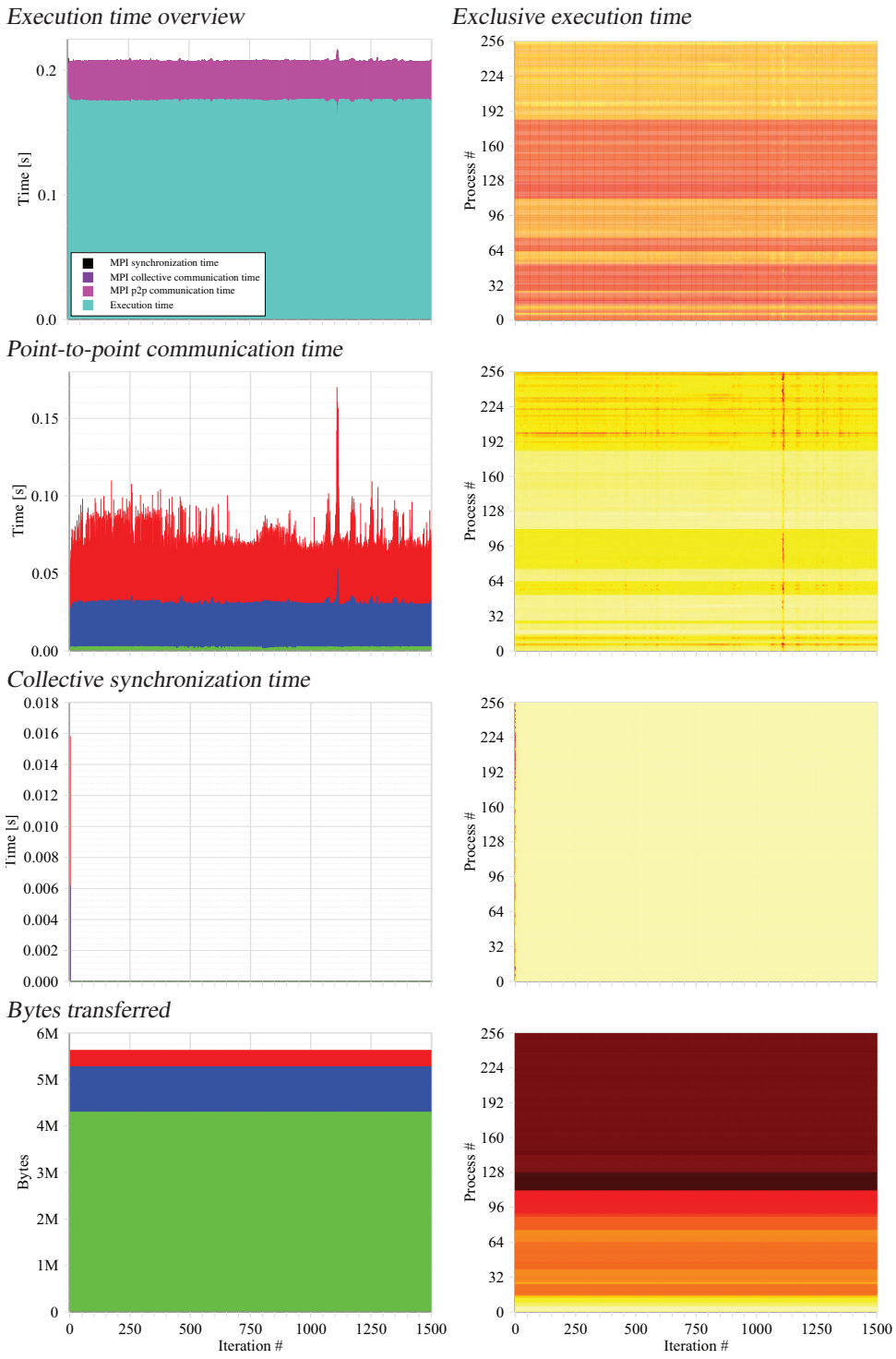


Figure B.13: *Iteration graphs and value maps of 145.lGemsFDTD.*

B.13 147.l2wrf2

147.l2wrf2 spends most of its time in useful computations. Its *Exclusive execution time* shows flat baseline behavior with periodic peaks every 40 iterations, where more calculations are included in the weather prediction model. The *Point-to-point communication time* shows a slight increase of the average over time with the minimum value staying flat, or even slightly decreasing. The periodic peaks are also present in this metric, at the same iterations. Both the relatively constant baseline of the *Exclusive execution time* and the constant *Communication count* suggest that the application's behavior does not change over time the same way as the *Point-to-point communication time*, there must be some other explanation for those changes.

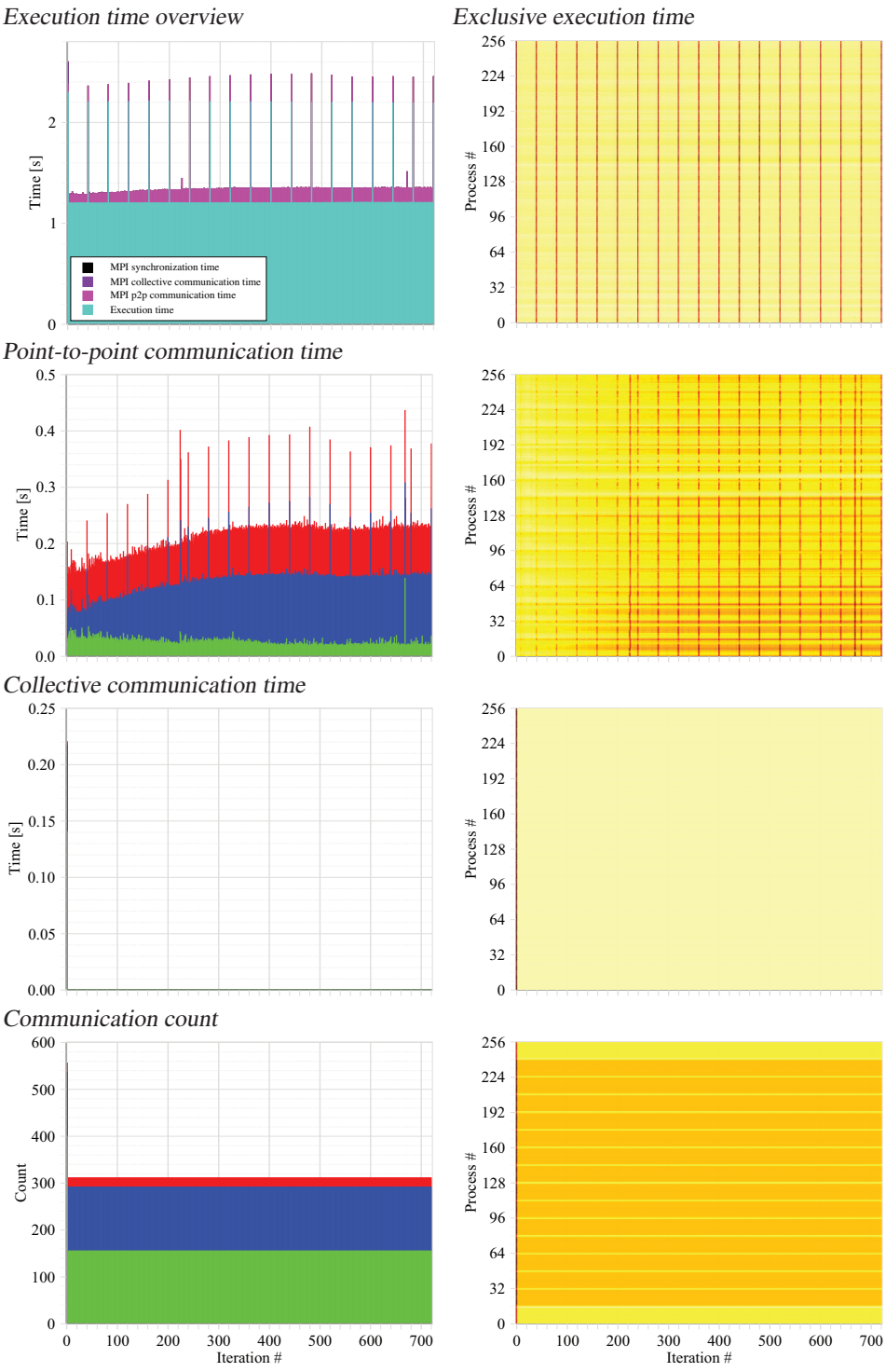


Figure B.14: *Iteration graphs and value maps of 147.l2wrf2.*

B.14 DROPS

The *Execution time overview* graph shows that *DROPS* spends around 80% of the time in useful computations, and its behavior changes relatively little over time. Still, the effect of the `reparam` function being called every 20th iteration is clearly visible: the next few iterations can work with a freshly optimized domain decomposition, which makes them slightly more efficient, but this advantage fades away over a few iterations as the simulated droplet moves. Due to the prohibitive measurement overhead associated with direct instrumentation of *DROPS*, this is a measurement based on the hybrid sampling method introduced in Chapter 5, and as such, we measure the measurement overhead of unwinding from MPI events, shown in red on the Execution time overview graph (there is no unwinding in the other measurements of this chapter, therefore the red color only appears in this section). The Exclusive execution time value map indicates that there is significant imbalance in the computation time, which causes some waiting time in both Point-to-point communication and synchronizing collective communication events. The *Point-to-point communication count* graph shows that there is also significant imbalance in the communication count, and also that the adaptive workload balancing causes the *Point-to-point communication count* to change considerably over time.

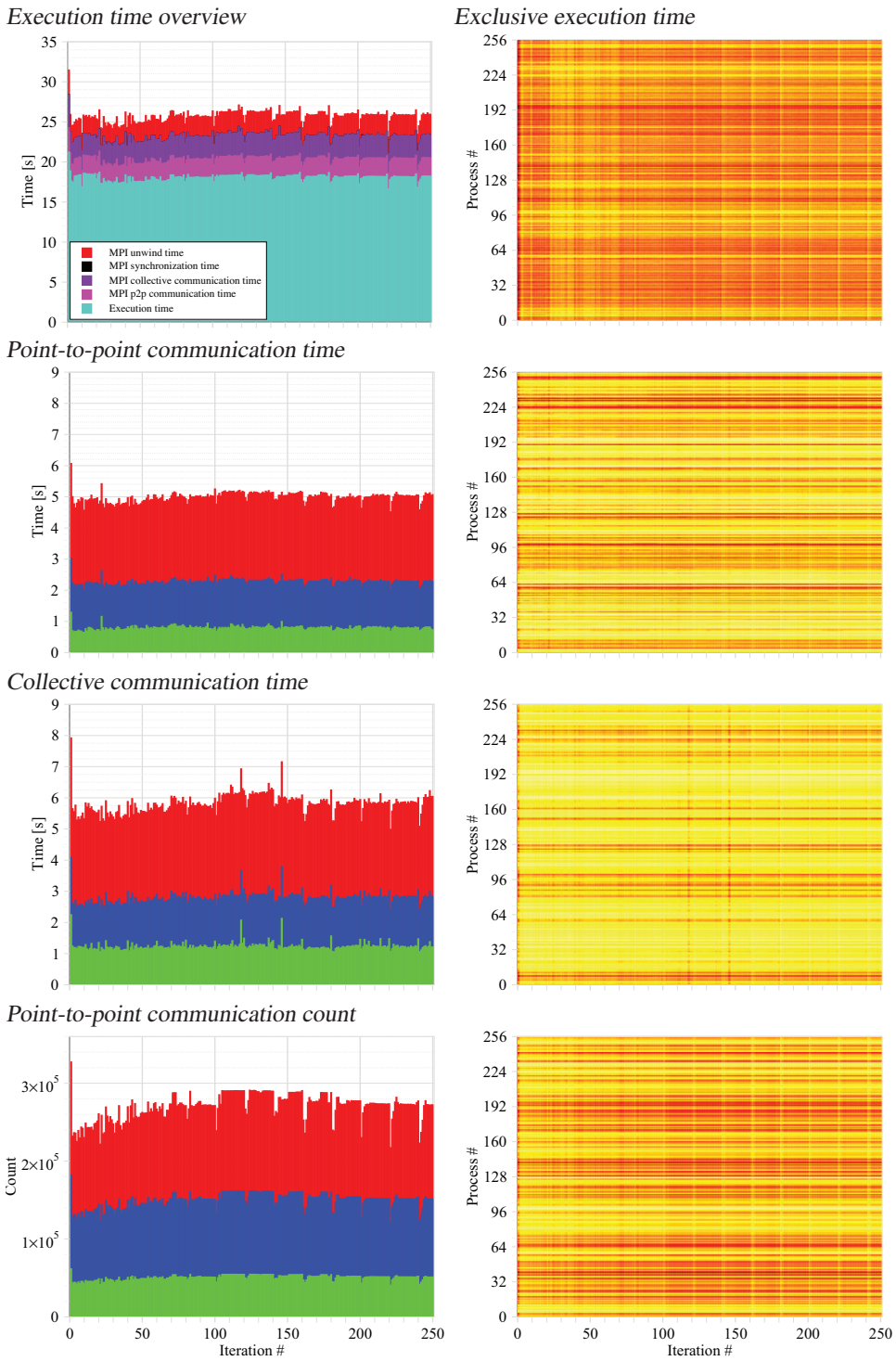


Figure B.15: *Iteration graphs and value maps of DROPS.*

B.15 PEPC

This analysis is based on a 1024-process measurement of *PEPC* on JUGENE (the measurements of all other codes were taken on JUROPA, but for *PEPC* JUGENE is the natural target platform). As we used a development version of the *PEPC* code for this measurement, it had many `MPI_Barrier` calls, mostly to separate the different main phases of the application for the developer's own profiling purposes. This lead to a significant portion of the time to be spent in *Collective synchronization time*, shown in black on the *Execution time overview* graph. As this is a tracing experiment, the peaks every 100th iteration are due to the fact that this measurement was collected in pieces of 100 iterations to avoid filling the trace buffer during measurement. Writing the output at these points also caused an increase in the *Collective communication time*. Still, the most important performance characteristics are the red lines in the *Point-to-point communication count* value maps, caused by too many particles being collected on a handful of processes by the workload balancing algorithm, as shown in Section 3.3. These processes become bottlenecks for the point-to-point communication, leading to processes with higher ranks having to wait much longer for certain messages, as seen on the *Point-to-point communication time* value map. This imbalance is later synchronized at barriers and synchronizing collective communication events, leading to even more waiting time. A more detailed version of this study is available in [85].

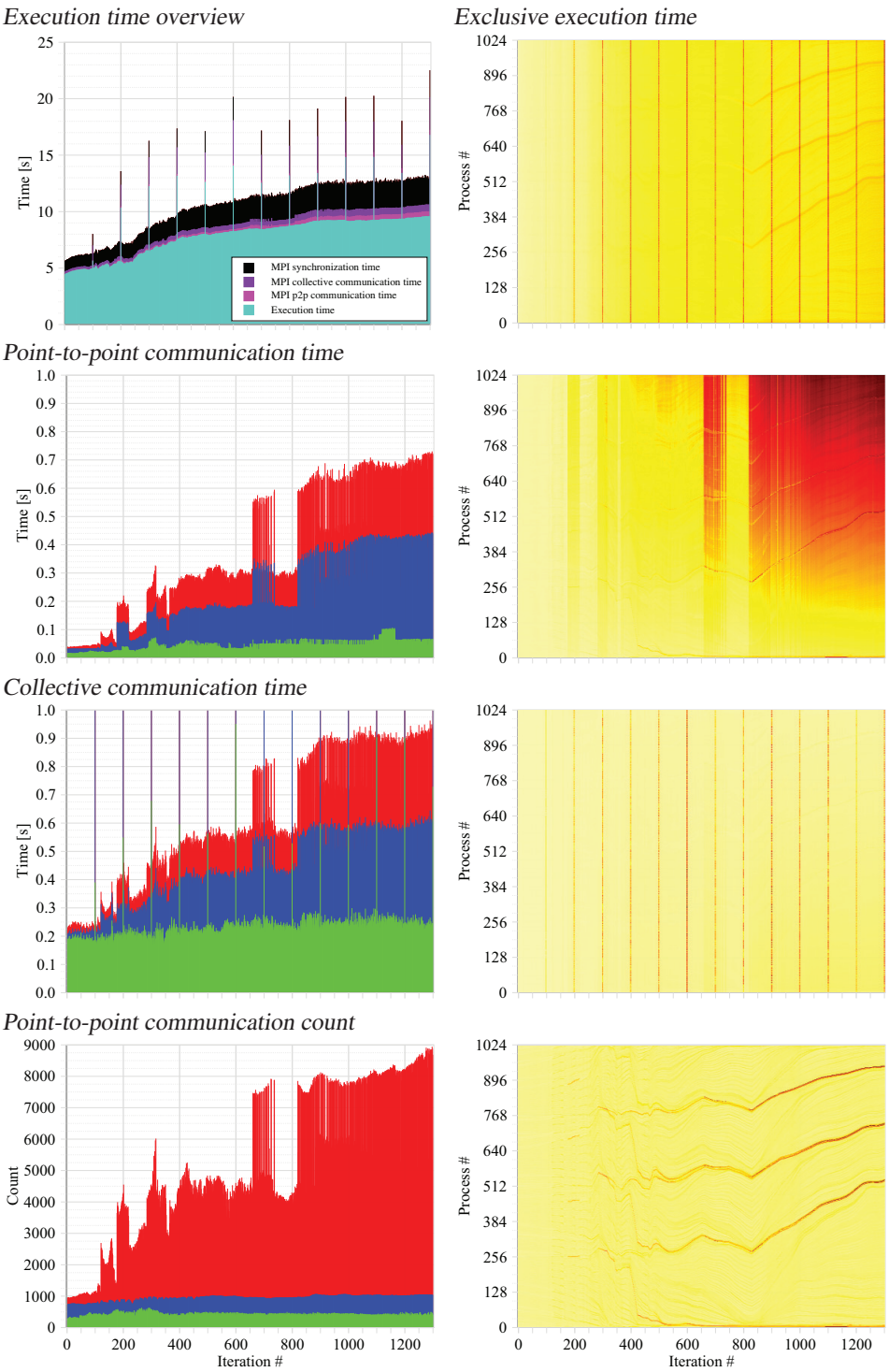


Figure B.16: *Iteration graphs and value maps of PEPC.*

References

[1] Green 500. The most powerful supercomputers ranked by energy efficiency. `http://www.green500.org`.

[2] Top 500. List of top 500 supercomputers. `http://top500.org`.

[3] Laksono Adhianto, Sinchan Banerjee, Michael W. Fagan, Mark W. Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, April 2010.

[4] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the Supercomputing Conference (SC2002), Baltimore, MD, USA, November 2002*. IEEE Computer Society, 2002.

[5] Ulf Andersson and Brian J. N. Wylie. Performance engineering of GemsFDTD computational electromagnetics solver. In *Proceedings of PARA 2010: State of the Art in Scientific and Parallel Computing, Reykjavík, Iceland, June 2010*. Springer, 2011 (to appear).

[6] Matthew Arnold and Peter F. Sweeney. Approximating the calling context tree via sampling. IBM Research Report 21789(98099), IBM Research Division, Almaden, CA, July 2000.

[7] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture (MICRO), Paris, France, December 1996*, pages 46–57. IEEE Computer Society, 1996.

[8] Robert Bell, Allen D. Malony, and Sameer S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), Klagenfurt, Austria, August 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2003.

[9] Andrew R. Bernat and Barton P. Miller. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience*, 19(11):1533–1547, August 2007.

[10] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA, September 2010*, pages 90–100. IEEE Computer Society, 2010.

[11] Holger Brunst and Wolfgang E. Nagel. Scalable performance analysis of parallel systems: Concepts and experiences. In *Proceedings of the 10th ParCo Conference, Dresden, Germany, September 2003*, volume 13 of *Advances in Parallel Computing*, pages 737–744. Elsevier, 2004.

[12] Marc Casas, Rosa M. Badia, and Jesús Labarta. Automatic phase detection of MPI applications. In *Proceedings of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany, September 2007*, volume 15 of *Advances in Parallel Computing*, pages 129–136. IOS Press, 2008.

[13] Anthony Chan, William Gropp, and Ewing Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[15] Cray Inc. Using Cray Performance Analysis Tools, 2010. `http://docs.cray.com/books/S-2376-51/`.

[16] Luiz A. DeRose and Felix Wolf. CATCH – A call-graph based automatic tool for capture of hardware performance metrics for MPI and OpenMP applications. In *Proceedings of the 8th Euro-Par Conference (Euro-Par), Paderborn, Germany, August 2002*, volume 2400 of *Lecture Notes in Computer Science*, pages 167–176. Springer, 2002.

[17] Dyninst Project. Stackwalker API, 2011. `http://www.paradyn.org/html/stackwalker1.1-features.html`.

[18] Jay Fenlason and Richard M. Stallman. *GNU prof - The GNU Profiler*. Free Software Foundation, Inc., 1997. `http://sourceware.org/binutils/docs/gprof/`.

[19] Oliver Fortmeier and H. Martin Bücker. A parallel strategy for a level set simulation of droplets moving in a liquid medium. In *Proceedings of VECPAR 2010, Berkeley, CA, USA, June 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2011.

[20] Nathan Froyd, John Mellor-Crummey, and Robert Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th International*

Conference on Supercomputing (ICS), Cambridge, MA, USA, June 2005, pages 81–90. ACM, 2005.

[21] Karl Fürlinger, Michael Gerndt, and Jack J. Dongarra. On using incremental profiling for the performance analysis of shared-memory parallel applications. In *Proceedings of the 13th International Euro-Par Conference, Rennes, France, August 2007*, volume 4641 of *Lecture Notes in Computer Science*, pages 62–71. Springer, 2007.

[22] Karl Fürlinger, Michael Gerndt, and Jack J. Dongarra. Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared-memory multiprocessors. In *Proceedings of 7th International conference on Computational Science, Beijing, China, May 2007*, volume 4488 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2007.

[23] Todd Gamblin, Robert Fowler, and Daniel A. Reed. Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS), Miami, FL, USA, April 2008*. IEEE Computer Society, 2008.

[24] Markus Geimer, Marc-André Hermanns, Christian Siebert, Felix Wolf, and Brian J. N. Wylie. Scaling performance tool MPI communicator management. In *Proceedings of the 18th European MPI Users' Group Meeting (EuroMPI), Santorini, Greece, September 2011*, volume 6960 of *Lecture Notes in Computer Science*, pages 178–187. Springer, 2011.

[25] Markus Geimer, Björn Kuhlmann, Farzona Pulatova, Felix Wolf, and Brian J. N. Wylie. Scalable collation and presentation of call-path profile data with CUBE. In *Proceedings of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany, September 2007*, volume 15 of *Advances in Parallel Computing*, pages 645–652. IOS Press, 2008.

[26] Markus Geimer, Pavel Saviankou, Alexandre Strube, Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. Further improving the scalability of the Scalasca toolset. In *Proceedings of the PARA 2010: State of the Art in Scientific and Parallel Computing, Minisymposium Scalable tools for High Performance Computing, Reykjavík, Iceland, June 2010*. Springer, 2011 (to appear).

[27] Markus Geimer, Sameer S. Shende, Allen D. Malony, and Felix Wolf. A generic and configurable source-code instrumentation component. In Gabrielle Allen, Jarek Nabrzyski, Ed Seidel, Geert Dick van Albada, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Proceedings of the International Conference on Computational Science (ICCS), Baton Rouge, LA, USA, May 2009*, volume 5545 of *Lecture Notes in Computer Science*, pages 696–705. Springer, 2009.

[28] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[29] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI), Bonn, Germany, September 2006*, volume 4192 of *Lecture Notes in Computer Science*, pages 303–312. Springer, 2006.

[30] Harald Servat Gelabert and Germán Llort Sánchez. Extrae user guide, 2010. `http://www.bsc.es/ssl/apps/performanceTools/files/docs/extrae-userguide.pdf`.

[31] Harald Servat Gelabert, Germán Llort Sánchez, Judit Giménez, Kevin A. Huck, and Jesús Labarta. Unveiling internal evolution of parallel application computation phases. In *Proceedings of the International Conference on Parallel Processing (ICPP), Taipei, Taiwan, September 2011*. IEEE Computer Society, 2011.

[32] Harald Servat Gelabert, Germán Llort Sánchez, Judit Giménez, and Jesús Labarta. Detailed performance analysis using coarse grain sampling. In *Euro-Par 2009 - Parallel Processing Workshops, Delft, The Netherlands, August 2009*, volume 6043 of *Lecture Notes in Computer Science*, pages 185–198. Springer, 2010.

[33] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. Periscope: Advanced techniques for performance analysis. In *Proceedings of the 11th ParCo Conference, Malaga, Spain, September 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 15–26. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[34] Paul Gibbon, Wolfgang Frings, Sonja Dominiczak, and Bernd Mohr. Performance analysis and visualization of the n-body tree code PEPC on massively parallel computers. In *Proceedings of the 11th ParCo Conference, Malaga, Spain, September 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 367–374. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[35] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, 1982.

[36] Sven Groß. *Numerical methods for three-dimensional incompressible two-phase flow problems*. PhD thesis, RWTH Aachen, 2008.

[37] Sven Groß, Jörg Peters, Volker Reichelt, and Arnold Reusken. The DROPS package for numerical simulations of incompressible flows using parallel adaptive multigrid techniques. IGPM-Report 211, RWTH Aachen University, 2002. `http://www.igpm.rwth-aachen.de/Download/reports/DROPS/IGPM211.pdf`.

[38] Sven Groß, Volker Reichelt, and Arnold Reusken. A finite element based level set method for two-phase incompressible flows. *Computing and Visualization in Science*, 9(4):239–257, 2004.

[39] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 2006.

[40] Torsten Hoefler, William Gropp, Rajeev Thakur, and Jesper Larsson Träff. Toward performance models of mpi implementations for understanding application scaling issues. In *Proceedings of the 17th European Message Passing Interface Conference (EuroMPI), Stuttgart, Germany, September 2010.*, volume 6305 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2010.

[41] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10), New Orleans, LA, USA, November 2010*. IEEE Computer Society, 2010.

[42] Jeffrey K. Hollingsworth, R. Bruce Irvin, and Barton P. Miller. The integration of application and system based metrics in a parallel program performance tool. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, Virginia, April 1991*, volume 26 of *SIGPLAN Notices*, pages 189–200. ACM, 1991.

[43] Kevin A. Huck and Allen D. Malony. PerfExplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the Supercomputing Conference (SC2005), Seattle, WA, USA, November 2005*. IEEE Computer Society, 2005.

[44] IBM. Blue Gene. *IBM Journal of Research and Development*, 49(2-3), 2005. `http://www.research.ibm.com/journal/rd49-23.html`.

[45] Intel. Trace Analyzer and Collector, 2011. `http://software.intel.com/en-us/intel-trace-analyzer/`.

[46] Intel. VTune Amplifier XE, 2011. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`.

[47] Marty Itzkowitz and Yukon Maruyama. HPC profiling with the Sun Studio Performance Tools. In *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, Dresden, Germany, September 2009*, pages 67–93. Springer, 2009.

[48] Christian Iwainsky and Dieter an Mey. Comparing the usability of performance analysis tools. In *Proceedings of the Workshop on Productivity and Performance (PROPER) in*

conjunction with Euro-Par 2008, Las Palmas de Gran Canaria, Spain, August 2008, volume 5415 of *Lecture Notes in Computer Science*, pages 315–325. Springer, 2009.

[49] Jülich Supercomputing Centre. JuRoPA – Jülich Research on Petaflop Architectures, 2010. `http://www.fz-juelich.de/jsc/juropa/`.

[50] Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Michael Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC2001), Denver, CO, USA, November 2001*, page 37, 2001.

[51] Darren J. Kerbyson, Kevin J. Barker, and Kei Davis. Analysis of the weather research and forecasting (WRF) model on large-scale systems. In *Proceedings of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany, September 2007*, volume 15 of *Advances in Parallel Computing*, pages 89–98. IOS Press, 2008.

[52] Andreas Knüpfer and Wolfgang E. Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. In *Proceedings of the International Conference on Parallel Processing (ICPP), Oslo, Norway, June 2005*, pages 165–172. IEEE Computer Society, 2005.

[53] Jesús Labarta, Judit Giménez, Eloy Martínez, Pedro González, Harald Servat Gelabert, Germán Llort Sánchez, and Xavier Aguilar. Scalability of tracing and visualization tools. In *Proceedings of the 11th ParCo Conference, Malaga, Spain September 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 869–876. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[54] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A parallel program development environment. In *Proceedings of the European Conference on Parallel Computing (Euro-Par), Lyon, France, August 1996*, volume 1124 of *Lecture Notes in Computer Science*, pages 665–674. Springer, 1996.

[55] Dmitry V. Levin, Roland McGrath, and Wichert Akkerman. strace, 2011. `http://strace.sourceforge.net/`.

[56] Los Alamos National Laboratory, Los Alamos, NM, USA. ASCI SWEEP3D v2.2b: Three-dimensional discrete ordinates neutron transport benchmark, 1995. `http://wwwc3.lanl.gov/pal/software/sweep3d/`.

[57] Charng-da Lu and Daniel A. Reed. Compact application signatures for parallel and distributed scientific codes. In *Proceedings of the Supercomputing Conference (SC2002), Baltimore, MD, USA, November 2002*. IEEE Computer Society, 2002.

[58] Allen D. Malony, Sameer S. Shende, and Alan Morris. Phase-based parallel performance profiling. In *Proceedings of the 11th ParCo Conference, Malaga, Spain, September 2005*,

volume 33 of *John von Neumann Institute for Computing Series*, pages 203–210. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[59] Karl Menger. Square circles (the taxicab geometry). In *Selected Papers in Logic and Foundations, Didactics, Economics*. D. Reidel Publishing Company, 1979.

[60] Message Passing Interface Forum. MPI: A message-passing interface standard, version 2.2: Profiling interface, September 2009. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

[61] Barton P. Miller, Morgan Clark, Jeffrey K. Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *Transactions on Parallel and Distributed Systems*, 1:206–217, 1990.

[62] Alan Morris, Allen D. Malony, Sameer S. Shende, and Kevin A. Huck. Design and implementation of a hybrid parallel performance measurement system. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP), San Diego, USA, September 2010*, pages 492–501. IEEE Computer Society, 2010.

[63] David Mosberger. libunwind, 2011. `http://www.nongnu.org/libunwind/`.

[64] Matthias S. Müller. Applying performance tools to real world applications. In *Proceedings of Seminar 07341 on Code Instrumentation for Massively Parallel Performance Analysis, Dagstuhl, Germany, August 2007*, 2007.

[65] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with Vampir, VampirServer and VampirTrace. In *Proceedings of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany, September 2007*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2008.

[66] Matthias S. Müller, G. Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, and Carl Ponder. SPEC MPI2007 — An application benchmark for clusters and HPC systems. In *Proceedings of the International Supercomputing Conference, Dresden, Germany, June 2007*, 2007.

[67] Jan Mußler, Daniel Lorenz, and Felix Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Proceedings of the European Conference on Parallel Computing (Euro-Par), Bordeaux, France, August 2011*, volume 6852 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2011.

[68] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12:69–80, 1996.

[69] Oleg Y. Nickolayev, Philip C. Roth, and Daniel A. Reed. Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, 1997.

[70] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69:696–710, 2009.

[71] OpenMP Architecture Review Board. OpenMP application program interface, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[72] RWTH Aachen University. Bullx B500 cluster, 2011. `http://top500.org/system/details/10813/`.

[73] Scalasca. Description of instrumentation/measurement regions, 2011. `http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_regions-1.3.html`.

[74] Scalasca. Scalable performance analysis of large-scale parallel applications, 2011. `http://www.scalasca.org`.

[75] Scalasca. Version 1.3 User Guide, 2011. `http://www2.fz-juelich.de/jsc/datapool/scalasca/UserGuide.pdf`.

[76] ScaleMP. Versatile SMP (vSMP) architecture. `http://www.scalemp.com/architecture/`.

[77] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2-3):105–121, 2008.

[78] Sameer S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, August 2001.

[79] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[80] Sameer S. Shende, Allen D. Malony, Alan Morris, Steven G. Parker, and John Davison de St. Germain. Performance evaluation of adaptive scientific applications using TAU. In *Proceedings of the International Conference on Parallel Computational Fluid Dynamics, Washington DC, USA, May 2005*, 2005.

[81] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, October 2002*, pages 45–57, 2002.

[82] Allan Snavely, Laura Carrington, Nicole Wolter, Jesús Labarta, Rosa M. Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the Supercomputing Conference (SC2002), Baltimore, MD, USA, November 2002*. IEEE Computer Society, 2002.

[83] Standard Performance Evaluation Corporation. SPEC MPI2007 benchmark suite, 2007. `http://www.spec.org/mpi2007/`.

[84] Zoltán Szebenyi, Todd Gamblin, Martin Schulz, Boris R. de Supinski, Felix Wolf, and Brian J. N. Wylie. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Anchorage, AK, USA, May 2011*, pages 637–648. IEEE Computer Society, 2011.

[85] Zoltán Szebenyi, Brian J. N. Wylie, and F. Wolf. Scalasca parallel performance analyses of PEPC. In *Proceedings of the Workshop on Productivity and Performance (PROPER) in conjunction with Euro-Par 2008, Las Palmas de Gran Canaria, Spain, August 2008*, volume 5415 of *Lecture Notes in Computer Science*, pages 305–314. Springer, 2009.

[86] Zoltán Szebenyi, Brian J. N. Wylie, and Felix Wolf. SCALASCA parallel performance analyses of SPEC MPI2007 applications. In *Proceedings of the 1st SPEC International Performance Evaluation Workshop, Darmstadt, Germany, June 2008*, volume 5119 of *Lecture Notes in Computer Science*, pages 99–123. Springer, 2008.

[87] Nathan R. Tallent, John Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, Dublin, Ireland, June 2009*, pages 441–452, New York, NY, USA, 2009. ACM.

[88] Jeffrey S. Vetter and Chris Chambreau. mpiP: Lightweight, scalable MPI profiling, 2011. `http://mpip.sourceforge.net/`.

[89] Jeffrey S. Vetter and Daniel A. Reed. Managing performance analysis with dynamical statistical projection pursuit. In *Proceedings of the Supercomputing Conference (SC1999), Portland, OR, USA, November 1999*. IEEE Computer Society, 1999.

[90] Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Wolfgang Frings, Karl Fürlinger, Markus Geimer, Marc-André Hermanns, Bernd Mohr, Shirley Moore, Matthias Pfeifer, and Zoltán Szebenyi. Usage of the Scalasca toolset for scalable

performance analysis of large-scale parallel applications. In *Proceedings of the 2nd International Workshop on Tools for High Performance Computing, Stuttgart, Germany, July 2008*, pages 157–167. Springer, 2008.

[91] Brian J. N. Wylie, Markus Geimer, Bernd Mohr, David Böhme, Zoltán Szebenyi, and Felix Wolf. Large-scale performance analysis of Sweep3D with the Scalasca toolset. *Parallel Processing Letters*, 20(4):397–414, December 2010.

[92] Brian J. N. Wylie and Darryl J. Gove. OMP AMMP analysis with Sun ONE Studio 8. In *Proceedings of the 5th European Workshop on OpenMP, Aachen, Germany*, pages 175–184, 2003.

[93] Brian J. N. Wylie, Bernd Mohr, and Felix Wolf. Holistic hardware counter performance analysis of parallel programs. In *Proceedings of the 11th ParCo Conference, Malaga, Spain, September 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 187–194. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[94] Brian J. N. Wylie, Felix Wolf, Bernd Mohr, and Markus Geimer. Integrated runtime measurement summarisation and selective event tracing for scalable parallel execution performance diagnosis. In *Proceedings of the Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umeå, Sweden, June 2006*, volume 4699 of *Lecture Notes in Computer Science*, pages 460–469. Springer, 2007.

Résumé

Personal data

Last name	Szebenyi
First name	Zoltán Péter
Date of birth	14 September 1983
Place of birth	Szeged, Hungary
Citizenship	Hungarian

Qualifications

1998 – 2002	Endre Ságvári High School, Szeged, Hungary Graduated in 2002
2002 – 2007	Computer science student, University of Szeged, Hungary Diploma in 2007
2006 – 2007	Erasmus exchange student at the Friedrich-Alexander-University of Erlangen-Nürnberg Erlangen, Germany
since 2007	Doctoral student at the Aachen Institute for Advanced Study in Computational Engineering Science, RWTH Aachen University, Germany
2007 – 2011	Guest scientist at the Jülich Supercomputing Centre, Jülich, Germany
2009	Five months internship at Lawrence Livermore National Laboratory, Livermore, CA, USA
since 2011	Scientific staff member at the German Research School for Simulation Sciences, Aachen, Germany

1. **Three-dimensional modelling of soil-plant interactions: Consistent coupling of soil and plant root systems**
   by T. Schröder (2009), VIII, 72 pages
   ISBN: 978-3-89336-576-0
   URN: urn:nbn:de:0001-00505

2. **Large-Scale Simulations of Error-Prone Quantum Computation Devices**
   by D. B. Trieu (2009), VI, 173 pages
   ISBN: 978-3-89336-601-9
   URN: urn:nbn:de:0001-00552

3. **NIC Symposium 2010**
   Proceedings, 24 – 25 February 2010 | Jülich, Germany
   edited by G. Münster, D. Wolf, M. Kremer (2010), V, 395 pages
   ISBN: 978-3-89336-606-4
   URN: urn:nbn:de:0001-2010020108

4. **Timestamp Synchronization of Concurrent Events**
   by D. Becker (2010), XVIII, 116 pages
   ISBN: 978-3-89336-625-5
   URN: urn:nbn:de:0001-2010051916

5. **UNICORE Summit 2010**
   Proceedings, 18 – 19 May 2010 | Jülich, Germany
   edited by A. Streit, M. Romberg, D. Mallmann (2010), iv, 123 pages
   ISBN: 978-3-89336-661-3
   URN: urn:nbn:de:0001-2010082304

6. **Fast Methods for Long-Range Interactions in Complex Systems**
   Lecture Notes, Summer School, 6 – 10 September 2010**,** Jülich, Germany
   edited by P. Gibbon, T. Lippert, G. Sutmann (2011), ii, 167 pages
   ISBN: 978-3-89336-714-6
   URN: urn:nbn:de:0001-2011051907

7. **Generalized Algebraic Kernels and Multipole Expansions for Massively Parallel Vortex Particle Methods**
   by R. Speck (2011), iv, 125 pages
   ISBN: 978-3-89336-733-7
   URN: urn:nbn:de:0001-2011083003

8. **From Computational Biophysics to Systems Biology (CBSB11)**
   Proceedings, 20 - 22 July 2011 | Jülich, Germany
   edited by P. Carloni, U. H. E. Hansmann, T. Lippert, J. H. Meinke, S. Mohanty, W. Nadler, O. Zimmermann (2011), v, 255 pages
   ISBN: 978-3-89336-748-1
   URN: urn:nbn:de:0001-2011112819

9. **UNICORE Summit 2011**
   Proceedings, 7 - 8 July 2011 | Toruń, Poland
   edited by M. Romberg, P. Bała, R. Müller-Pfefferkorn, D. Mallmann (2011), iv, 150 pages
   ISBN: 978-3-89336-750-4
   URN: urn:nbn:de:0001-2011120103

10. **Hierarchical Methods for Dynamics in Complex Molecular Systems**
    Lecture Notes, IAS Winter School, 5 – 9 March 2012, Jülich, Germany
    edited by J. Grotendorst, G. Sutmann, G. Gompper, D. Marx (2012), vi, 540 pages
    ISBN: 978-3-89336-768-9
    URN: urn:nbn:de:0001-2012020208

11. **Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method**
    by I. Kabadshow (2012), v, 126 pages
    ISBN: 978-3-89336-770-2
    URN: urn:nbn:de:0001-2012020810

12. **Capturing Parallel Performance Dynamics**
    by Z. P. Szebenyi (2012), xxi, 192 pages
    ISBN: 978-3-89336-798-6
    URN: urn:nbn:de:0001-2012062204

Runtime call-path profiling is a conventional, well-known method used for collecting summary statistics of a parallel application's execution such as the time spent in different call paths of the code. However, these kinds of measurements give the user only a summary overview of the entire execution, without regard to changes in performance behavior over time. As present day scientific applications tend to be run for extended periods of time, understanding the patterns and trends in the performance data along the time axis becomes crucial.

As shown by our analysis of a representative set of scientific codes, profiling every iteration separately provides a wealth of new data that often leads to invaluable new insights. However, with the introduction of the time dimension, memory usage and file sizes grow considerably. To counter this problem, a low-overhead on-line compression algorithm was developed that exploits similarities between different iterations.

While standard, direct instrumentation, which is assumed by the initial version of the compression algorithm, results in fairly low overhead with many scientific codes, in some cases the high frequency of events makes such measurements impractical. Therefore, a hybrid solution was developed that seamlessly integrates sampling and direct instrumentation techniques in a single unified measurement, using direct instrumentation for message passing constructs, while sampling the rest of the code. Finally, the compression algorithm was adapted to the hybrid profiling approach, avoiding the overhead of pure direct instrumentation.

Evaluation of the above methodologies shows that our similarity-based compression algorithm provides a very good approximation of the original data with very little measurement dilation, while the hybrid combination of sampling and direct instrumentation fulfills its purpose by showing the expected reduction of measurement dilation in cases unsuitable for direct instrumentation.

This publication was written at the Jülich Supercomputing Centre (JSC) which is an integral part of the Institute for Advanced Simulation (IAS). The IAS combines the Jülich simulation sciences and the supercomputer facility in one organizational unit. It includes those parts of the scientific institutes at Forschungszentrum Jülich which use simulation on supercomputers as their main research methodology.

IAS Series
Volume 12
ISBN 978-3-89336-798-6

JÜLICH
FORSCHUNGSZENTRUM